collection DataPro

Algorithmique

Raisonner pour concevoir

Christophe HARO

eni éditions

Algorithmique

Raisonner pour concevoir

Christophe HARO

Résumé

Ce livre sur **l'algorithmique** est destiné à toute personne qui s'intéresse au développement d'applications informatiques et qui souhaite s'initier ou retrouver les **bases fondamentales de la programmation**. Il ne s'agit pas ici de programmer avec un langage ou un autre, mais bien de raisonner sur un problème pour concevoir une solution abstraite. Ce **travail de réflexion et de conception** prépare le stade ultime de l'implémentation et du cycle de vie du programme concret.

Le lecteur ne trouvera pas dans ce livre un recueil d'algorithmes qu'il devra ensuite adapter pour résoudre des problèmes mais au contraire une **introduction originale et efficace à l'algorithmique** pour apprendre à analyser un problème.

Le livre est divisé en deux parties. Dans la première partie sont détaillées les notions **d'algorithmique de base** et la **méthode de construction raisonnée** d'un algorithme impératif : l'auteur y précise notamment la distinction entre la spécification et la réalisation d'un algorithme et montre que l'algorithmique proprement dite s'arrête là où commence la programmation. Dans la deuxième partie l'auteur propose cette fois des **solutions à des problèmes** plus élaborés dans divers domaines du calcul automatique, comme la simulation de phénomènes aléatoires ou le cryptage des données.

Toutes les activités proposées restent élémentaires avec le souci constant de **privilégier le raisonnement** qui conduit à l'élaboration des algorithmes.

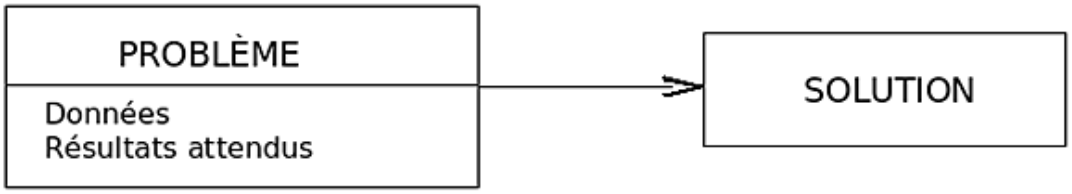
L'auteur

Ingénieur et docteur en informatique, **Christophe Haro** a enseigné l'informatique à l'université et en école d'ingénieurs pendant 22 ans. Depuis près de 10 ans il enseigne le génie logiciel, le développement d'applications informatiques et les architectures logicielles en BTS Informatique de Gestion. C'est toute cette expérience pédagogique qui donne à ce livre son efficacité pour aborder l'algorithmique.

Ce livre numérique a été conçu et est diffusé dans le respect des droits d'auteur. Toutes les marques citées ont été déposées par leur éditeur respectif. La loi du 11 Mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayant cause, est illicite" (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal. Copyright Editions ENI

Qu’est-ce que l’algorithmique ?

On s’intéresse à l’activité de programmation d’un ordinateur qui permet de résoudre des problèmes d’une façon automatique. On se place à un niveau conceptuel dans lequel un problème quelconque, informatique ou non, est caractérisé par un ensemble de données d’entrée et des résultats attendus. On peut donc représenter ce niveau d’analyse par le schéma de la figure ci-dessous.



Exemple

On donne les ingrédients suivants :

Thym - laurier - persil - ail - huile - vinaigre - sel - poivre - citron et un brochet de trois livres. Préparer une marinade de brochet.

Dans cet exemple, le problème est représenté par les données : le thym, le laurier, ..., le brochet dont on connaît le poids et par le résultat attendu qui est un plat cuisiné : une marinade de brochet. Ce n’est pas (pas encore) un problème qui se pose habituellement en informatique, mais il se pose dans les mêmes termes.

Exemple

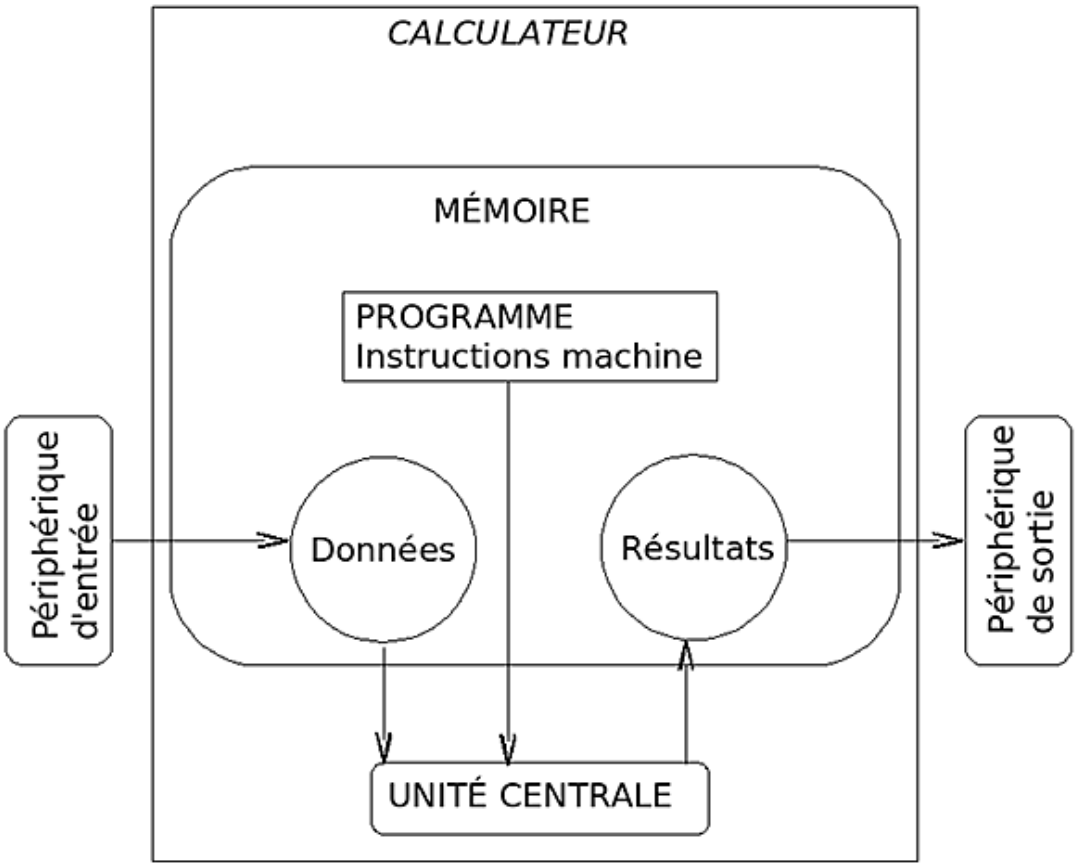
On demande de calculer, pour une année dont on donne le millésime, le jour de la semaine où tombe le premier mai.

Ici, les données sont constituées d’un millésime, comme 2005 par exemple. Le résultat attendu est un jour de la semaine, comme dimanche par exemple.

Dans ces exemples, le problème est constitué d’un jeu de données. Le résultat attendu doit être obtenu par des transformations à faire subir aux données ; c’est ce que représente la figure ci-dessus.

Les problèmes auxquels on s’intéresse dans ce livre sont résolus à l’aide d’un ordinateur. Un ordinateur est constitué d’une machine matérielle complétée d’un ensemble de composants périphériques, comme un clavier, une souris, une table traçante... La machine matérielle est constituée, en première approximation, d’une unité centrale, chargée d’effectuer les calculs, complétée d’une mémoire, permettant d’enregistrer les données, les résultats des calculs intermédiaires et les résultats attendus. Pour résoudre un problème avec l’aide d’un ordinateur, il faut enregistrer dans sa mémoire, d’une part les données du problème, qu’il devra transformer pour obtenir le résultat attendu et, d’autre part, la suite des instructions que l’unité centrale du calculateur devra exécuter pour effectuer les transformations des données. Le résultat étant obtenu, il sera d’abord enregistré, comme les données, dans la mémoire. La figure suivante représente ces éléments.

© ENI Editions - All rigths reserved



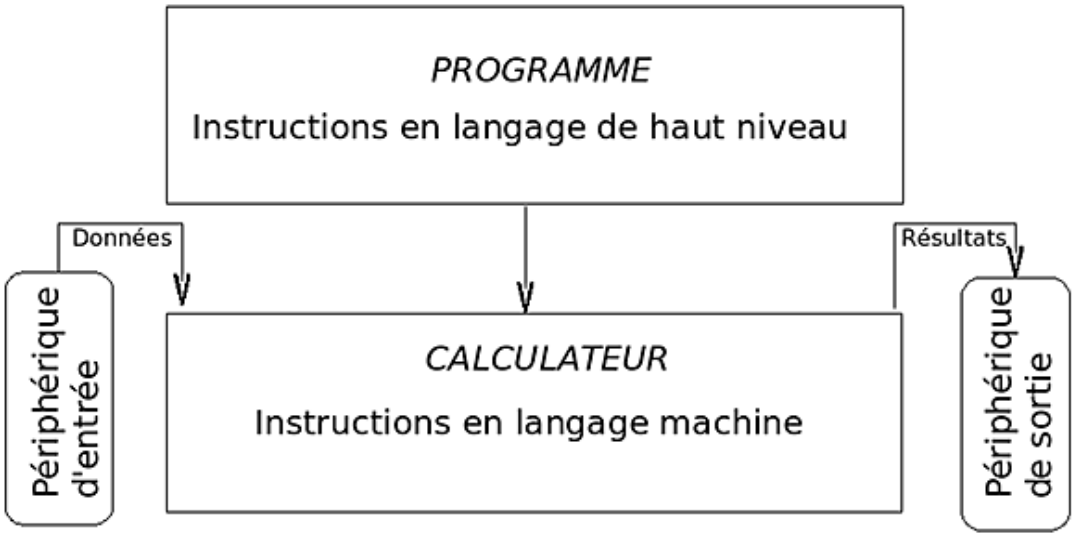
Un périphérique d'entrée permet de communiquer au programme les données à transformer. Un programme, constitué d'instructions en langage machine exécutées par l'unité centrale, les transforme et produit des résultats qui sont envoyés au périphérique de sortie.

Un premier élément, remarquable ici, est l'architecture mise en œuvre. Les instructions machine exécutées par l'unité centrale et les données sur lesquelles elles agissent sont rangées et cohabitent dans la même mémoire centrale. Une telle architecture est celle des ordinateurs depuis leur tout début, dans la première moitié du vingtième siècle. Elle a été inventée et théorisée par John VON NEUMANN et Alan THÜRING. Dans cette architecture, l'unité centrale n'exécute qu'une seule instruction à la fois et ces instructions sont exécutées séquentiellement, l'une après l'autre, dans un ordre préétabli et définitivement fixé. On dit qu'il s'agit d'une *machine séquentielle* pour caractériser un tel comportement. Il existe d'autres types de machines qui ne nous intéressent pas dans ce livre.

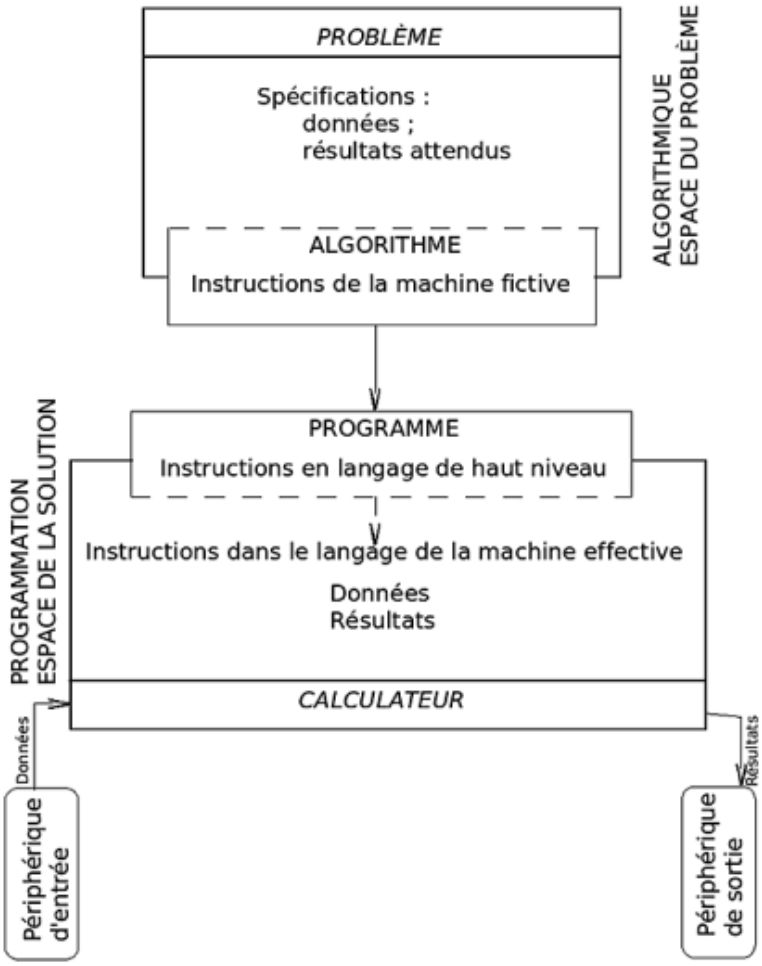
Une deuxième propriété importante de telles machines est que les mêmes données, communiquées au même programme, produisent les mêmes résultats, indépendamment de l'environnement et du moment de l'exécution des instructions. On dit alors que ce type de machine est *déterministe*. Là encore, ce n'est pas la seule architecture matérielle possible, mais ce livre se restreint à ce cas.

Pour obtenir un programme et les instructions qui le composent, on le prépare en utilisant un langage de programmation à l'aide duquel on écrit les instructions qui transforment les données. Il existe de nombreux langages de programmation de haut niveau, certains dédiés à des tâches spécifiques, comme les tableurs par exemple, d'autres destinés à écrire des programmes plus généraux, comme JAVA, PHP, C... La figure ci-dessous représente les relations entre un programme de haut niveau et le calculateur auquel il est destiné.

© ENI Editions - All rigths reserved



Pour passer des données d'un problème aux résultats attendus, on considère une machine fictive caractérisée par un comportement qui lui permet de réaliser les opérations. Cette machine fictive, comme son modèle, l'ordinateur de VON NEUMANN, n'effectue qu'une seule opération à la fois pour réaliser un traitement précisément défini. Les opérations s'organisent en une suite ordonnées d'instructions qui opèrent sur de l'information symbolique, c'est-à-dire sur une suite de signes alphabétiques, numériques, lexicaux. Cette machine fictive est donc une machine séquentielle, comme son modèle. On appelle *calcul* une suite de traitements réalisés par cette machine, *machine abstraite* le calculateur fictif et *programme abstrait* la suite d'instructions préparées pour une telle machine. Cependant, alors que les programmes écrits dans un langage de programmation et destinés à un calculateur concret doivent respecter une syntaxe précise et rigide, les programmes abstraits ne sont contraints par aucune règle de « grammaire », aucune syntaxe imposée. En ce sens, un programme abstrait n'est pas un programme et n'utilise pas un langage différent du langage naturel. On appelle *algorithme* un programme écrit pour la machine abstraite. La figure ci-dessous représente les relations entre la machine abstraite et son modèle, le calculateur concret.



© ENI Editions - All rigths reserved

Tout ordinateur et son langage sont donc la réalisation particulière d'une machine abstraite. Cependant, alors que le programme concret, destiné à un calculateur concret, appartient à l'espace de la solution du problème que nous voulons résoudre, la machine abstraite et son langage, l'algorithme, appartiennent à l'espace du problème à résoudre. La figure ci-dessus montre les différents espaces d'intervention. L'algorithme est le « langage » de la machine fictive. On y reconnaît deux domaines indépendants et bien distincts :

- la *spécification du problème* qui le définit entièrement et d'une façon non ambiguë. C'est le domaine des Mathématiques et de la logique.

- Les *instructions de la machine fictive* est un domaine qui assure la transition vers les instructions en langage de haut niveau destinées à la machine effective.

Dans l'espace de la solution, le langage de haut niveau et le langage machine réalisent les instructions de la machine effective. Pour être plus précis, il faudrait considérer que le langage de haut niveau introduit une machine fictive intermédiaire, mais cela ne sera pas fait ici pour simplifier.

© ENI Editions - All rigths reserved

Structure du livre

Ce livre ne s'intéresse qu'à la machine abstraite, dans l'espace du problème, pour laquelle il étudie une algorithmique particulière : l'algorithmique impérative ou procédurale. Il est divisé en deux parties. La première regroupe les chapitres 2 à 7. C'est dans cette partie que sont abordées les notions d'algorithmique de base et la méthode de construction raisonnée d'un algorithme impératif. La deuxième partie s'étend des chapitres 8 à 12. On y cherche, cette fois, des solutions à des problèmes plus élaborés dans divers domaines du calcul automatique.

Le chapitre 2, Programmes directs montre quelques exemples d'algorithmes, empruntés au quotidien, et tente de définir la nature des algorithmes étudiés dans cet ouvrage. On y précise, notamment, la distinction entre la spécification et la réalisation d'un algorithme et on montre que l'algorithmique proprement dite s'arrête là où commence la programmation. En particulier, résoudre un problème consiste à passer des données aux résultats attendus par la suite séquentielle des transformations à faire subir aux données. Cependant, la description de ces opérations peut s'exprimer de différentes façons et, en ce sens, l'algorithmique n'est pas la programmation. On aborde les notations qui seront utilisées pour exprimer des transformations sans contrôle du flot d'instructions.

Le chapitre 3, L'alternative étudie l'une des constructions fondamentales de l'algorithmique : l'alternative. Elle permet de choisir les traitements à réaliser en fonction de conditions sur les données, exprimées à l'aide de prédicats de la logique booléenne. C'est la première structure étudiée pour contrôler le flot d'instructions. De nombreux exercices permettent de la mettre en pratique.

Le chapitre 4 Structures élémentaires débute l'étude des chaînes de caractères et des tableaux statiques simples (vecteurs). On commence à préparer et à utiliser quelques définitions d'algorithmes élaborés qui interviennent sur ces structures de données. Le chapitre montre ensuite comment définir de nouveaux types de données à partir des types de base et propose quelques exercices d'application. Cependant, ce livre ne traite pas des structures de données. On n'y trouve pas d'étude des listes, files, arbres... De nombreux livres, certains excellents, étudient ces domaines en détail. Ici, seuls les chaînes de caractères et les tableaux statiques sont utilisés, mais comme support, comme prétexte à la définition raisonnée d'algorithmes élémentaires. L'objectif du livre est le *raisonnement* à la base de la construction d'algorithmes et non pas la solution aux problèmes génériques de la programmation informatique.

Nous verrons dans le chapitre 5 Récursivité que l'on peut ainsi spécifier un algorithme d'une façon opératoire, sans formalisme mathématique excessif. De plus, on peut alors distinguer plus clairement spécification et réalisation.

Le chapitre 6 Itération est fondamental. On y apprend une méthode systématique de construction des itérations. On montre à l'aide d'exemples variés qu'une itération est l'expression algorithmique d'une récurrence mathématique. L'hypothèse de récurrence permet de construire le corps de l'itération et de définir sa propriété invariante. La récursivité est utilisée pour la spécification des algorithmes itératifs.

Le chapitre 7 Récursivité ou Itération ? complète les chapitres Récursivité et Itération.

Le chapitre 8 Trier aborde un vaste sujet, le tri de données comparables, par quelques exemples usuels du domaine. On étudie avec soin la spécification d'un tri. C'est un problème général difficile, mais les algorithmes de tri présentés sont ceux habituellement étudiés dans une initiation à l'algorithmique.

Les chapitres précédents ont introduit à plusieurs reprises le problème de l'édition d'un nombre qui consiste à obtenir une chaîne de caractères représentant sa valeur dans une base quelconque. Le chapitre 9 Édition d'un nombre réalise une synthèse de ce sujet en résolvant entièrement les problèmes déjà rencontrés dans les chapitres précédents. Il étudie également un petit cas de gestion concernant l'identification des entreprises et de leurs établissements. Il propose aussi la résolution de problèmes de même nature, comme l'identification ISBN pour les livres par exemple.

Les premières notions sur les fichiers sont abordées au chapitre 10 Introduction aux fichiers. L'étude reste élémentaire. Elle présente les organisations séquentielles et adressables puis les modes d'accès associés. L'accent est mis sur les traitements des fichiers à accès séquentiel.

Le chapitre 11 Simuler montre comment « simuler le hasard ». Des exemples simples permettent d'introduire la simulation de phénomènes déterministes aussi bien qu'aléatoires.

Le chapitre 12 Crypter est encore un prétexte à l'écriture d'algorithmes simples, mais sur des exemples empruntés à la cryptographie cette fois. Là encore, le sujet concerne les raisonnements étudiés dans les chapitres précédents et les notions abordées restent triviales.

On ne trouvera quasiment jamais, dans ce livre, de code écrit dans un langage de programmation. Encore une fois, il ne s'agit pas ici de programmer, mais bien de raisonner sur un problème pour en concevoir une solution abstraite. Cependant, il convient de ne jamais perdre de vue que l'objectif est bien d'obtenir, *in fine*, un programme exécutable sur un calculateur concret. Même si, dans ce livre, on s'arrête là où commence la programmation, ce travail de réflexion et de conception prépare le stade ultime de l'implémentation et du cycle de vie du programme concret. Redescendre ainsi « sur Terre » imposera parfois des ajustements pour adapter l'algorithme, c'est-à-dire le programme qui cible la machine abstraite, aux contingences matérielles des machines concrètes et des langages de programmation concrets qui sont à la disposition du développeur. Anticipons donc pour illustrer ce point d'un exemple simple.

En algorithmique, le calcul de la moyenne arithmétique m de deux entiers a et b s'écrit comme en Mathématique : m <- (a+b)/2 (dans lequel <- se lit « prend la valeur »). Ainsi, m prend la valeur de la demi-somme de a et b. Cependant, le problème informatique concret ne se résout pas si facilement. En effet, la représentation des entiers dans la mémoire des calculateurs concrets contraint ces entiers à un domaine restreint de valeurs. Sur un calculateur qui utilise des mots mémoire de 32 bits par exemple, les entiers positifs supérieurs strictement à 2^{31}-1 ne peuvent pas être représentés (en arithmétique signée et en simple précision, mais ignorons ces détails pour rester simple). Ainsi, on ne

© ENI Editions - All rigths reserved

peut pas transposer directement ce calcul sur un ordinateur concret sans s'interroger sur le domaine des valeurs des opérandes a et b. Et si la somme a+b peut dépasser la valeur maximum d'un entier représentable en machine, il faudra bien, à un moment ou à un autre, étudier plus précisément le problème d'implémentation en tenant compte à la fois du calculateur cible et du langage de programmation utilisés pour résoudre le problème. C'est pourquoi on trouvera parfois un exemple d'implémentation dans un langage particulier de tel ou tel algorithme. Les langages utilisés seront toujours le langage C ou l'un de ses dérivés, comme PHP. Ces exemples ne seront que des digressions qui pourront être ignorées en première lecture.

© ENI Editions - All rigths reserved

Public visé

Ce livre s'adresse d'abord aux débutants en programmation. À ce titre, il est plutôt destiné aux filières qui enseignent encore la programmation impérative procédurale, comme les Sections de Techniciens Supérieurs en informatique (STS) ou les formations aux Diplômes Universitaires de Technologie (DUT). Cependant, les méthodes abordées et les exemples étudiés peuvent intéresser tout étudiant qui a à connaître ou à pratiquer la programmation en général. Les élèves ingénieurs trouveront certainement de multiples sujets d'analyse à leur niveau, peut-être en étendant certains des exercices proposés.

Les étudiants déjà avancés pourront aborder ce livre en piochant, ici ou là, des exercices à résoudre. Cependant, ils devront commencer par rafraîchir leurs connaissances en revoyant d'abord attentivement le chapitre Programmes directs qui traite de la spécification et le chapitre Itération qui traite de la construction d'une itération. Les débutants devront probablement étudier le livre linéairement, en suivant l'ordre des chapitres, au moins dans la première partie qui s'étend des chapitres Programmes directs à Récursivité ou Itération ? Les chapitres de la deuxième partie peuvent être abordés dans un ordre quelconque, mais il semble judicieux de commencer l'étude de cette partie par le chapitre Trier qui traite des algorithmes de tri élémentaires.

Quel que soit le niveau du lecteur qui entreprend l'aventure, il convient de garder à l'esprit que ce livre ne se lit pas comme un roman. Le sujet abordé est difficile et cette difficulté n'est pas la conséquence des choix adoptés, mais bien plutôt une difficulté inhérente au domaine étudié. Par conséquent, il semble nécessaire d'aborder les différentes notions avec un papier, un crayon et une bonne gomme. Les exercices devraient être résolus avant d'en lire la solution, ne serait-ce que pour bien se convaincre qu'il n'existe pas qu'une seule solution à un problème donné, mais aussi que les notations adoptées ne sont pas les seules et que l'algorithmique ne se réduit pas au langage utilisé pour exprimer un algorithme. Accessoirement, il sera peut-être utile, dans certains cas, d'aller jusqu'à l'implémentation d'un algorithme dans un langage de programmation particulier pour une machine concrète donnée. Cependant, il faudra toujours garder à l'esprit que si l'implémentation peut nous convaincre de ce qu'une solution est incorrecte, elle ne permet jamais de prouver qu'une solution est correcte.

© ENI Editions - All rigths reserved

Conventions adoptées

Le texte est rédigé selon certaines conventions typographiques qui sont présentées ici.

Signale une remarque importante ou un point particulier à noter.

Une remarque peut apparaître partout dans le texte, sauf dans les instructions des algorithmes présentés.

Exemple

est utilisé pour introduire un exemple qui illustre une notion ou un concept.

Les algorithmes et les exercices sont distingués par une présentation encadrée, comme ceci :

Algorithme 1 : Division euclidienne de deux entiers : version 2

Texte de définition de l'algorithme

Suite du texte de définition

Les extraits de code sont mis en valeur dans `cette police` de caractères.

Les références bibliographiques sont précisées dans le texte par des abréviations entre crochets, exemple [ARS80]. Pour les interpréter, vous pouvez vous reporter au titre Bibliographie en fin de chaque chapitre.

Il est temps de commencer.

© ENI Editions - All rigths reserved

Introduction

Ce chapitre présente la notion d'algorithme comme la description d'un procédé composé d'une succession d'opérations. À partir d'exemples simples tirés de l'expérience quotidienne de chacun et pas nécessairement dans le domaine de l'informatique, on dégage une idée intuitive de ce qu'est un algorithme et on commence à en écrire d'une façon informelle. Une série d'exercices permet ensuite de s'entraîner à élaborer des descriptions dans des domaines variés.

La section Premiers exemples introduit quelques exemples. La section suivante définit informellement un algorithme comme une suite d'instructions destinées à la machine abstraite. On peut alors commencer à introduire les spécifications dans la section Définition informelle d'un algorithme et écrire les premiers algorithmes à la section Spécifications. Les deux sections suivantes sont des exercices. Ils sont accompagnés d'une solution développée dans la section Exercices résolus et sont énoncés sans solution dans la section Exercice. On termine le chapitre par quelques notes bibliographiques.

© ENI Editions - All rigths reserved

Premiers exemples

Exemple 1

Dans un vieux livre de cuisine, je lis la recette suivante, en page 41 :

« MARINADE

Thym - laurier - persil - ail - huile - vinaigre - sel - poivre.

Mettez dans un plat 2 volumes d'huile, pour 1 de vinaigre, le thym, laurier, persil, l'ail haché finement, salez, poivrez et faites baigner dans cette préparation la viande que vous voulez faire mariner. Vous pouvez laisser mariner 2 à 6 jours, mais chaque jour ajouter de l'huile et du vinaigre. Pour le poisson, vous pouvez remplacer le vinaigre par du jus de citron. »

Cette recette décrit un algorithme : *comment réaliser une marinade ?* À partir d'ingrédients précisés, il décrit une succession d'opérations qui les utilisent pour obtenir un résultat. Il pourrait énumérer les différentes opérations en les ordonnant :

1. mélanger dans un plat 2 volumes d'huile à 1 volume de vinaigre ; pour du poisson, remplacer le vinaigre par du jus de citron ;

2. hacher finement l'ail ;

3. ajouter à la vinaigrette le thym, le laurier, le persil et l'ail haché ;

4. saler et poivrer ;

5. placer la viande à faire mariner dans cette préparation ;

6. pendant 2 à 6 jours, ajouter chaque jour de l'huile et du vinaigre.

Exemple 2

Tout élève apprend à additionner deux nombres entiers positifs en classe primaire. Pour calculer la somme de 127 et 35, il apprend à « poser » l'addition, en plaçant en colonnes les chiffres des deux nombres, les unités du second sous les unités du premier, les dizaines du second sous les dizaines du premier, etc. Ceci fait, il utilise un répertoire de connaissances acquises préalablement, pour additionner les deux nombres chiffre à chiffre. Il doit donc savoir que 7+5 font 12, 2+3 font 5, etc. Il apprend aussi à reporter la retenue lorsque l'addition de deux chiffres donne une somme supérieure à 9. Il est aussi possible de détailler les étapes du calcul de la façon suivante :

1. poser 127 ;

2. poser 35 sous 127, avec 5 sous 7, 3 sous 2 ;

3. additionner 7 et 5 qui font 12 : poser 2 pour les unités du résultat et 1 de retenue pour les dizaines ;

4. additionner 2 et 3 qui font 5 pour les dizaines et augmenter le résultat de la retenue précédente : poser 6 ;

5. additionner les centaines 1 à 0 : poser 1 ;

6. le résultat est 162.

Cette description est comparable à la recette précédente. On considère les ingrédients que sont les nombres à additionner, 127 et 35. À partir des tables d'addition, dont la connaissance est un préalable, on applique un ensemble de règles qui transforme les données, pour obtenir un résultat, ici leur somme 162.

Exemple 3

Voici, toujours dans le livre de cuisine du premier exemple ci-dessus, à la page 303 cette fois, ce que l'on peut y lire :

« SAINT-HONORÉ

Pâte brisée (page 290) - pâte à choux (page 289) - caramel (page 265) - crème Chantilly (page 267).

Étendez la pâte brisée en forme de galette de 20 cm environ de diamètre. Posez-la sur une tôle beurrée. Faire cuire 30 minutes à four chaud.

Avec de la pâte à choux faites une dizaine de petits choux que vous faites cuire à four modéré 20 minutes. Après parfaite cuisson, glacez ces choux avec un caramel blond. Laissez refroidir. Cette préparation terminée garnissez votre galette avec les petits choux que vous collez avec le caramel chaud. Enfin, remplissez votre gâteau avec une crème Chantilly. »

On trouve alors, aux pages indiquées en tête de recette, les recettes particulières pour la pâte brisée, la pâte à choux...

Exemple 4

© ENI Editions - All rigths reserved

Le service du magasin d'une entreprise souhaite calculer automatiquement des propositions de réapprovisionnement pour les articles allant bientôt manquer. Pour cela, on dispose pour chaque article des informations suivantes :

- *la quantité actuelle en stock ;*
- *le délai moyen de réapprovisionnement ;*
- *des statistiques de vente sur les 12 derniers mois.*

On demande de décrire les calculs à effectuer pour déterminer dans quels cas le réapprovisionnement d'un produit doit être proposé.

On appelle *niveau de déclenchement* la quantité d'un produit en deçà de laquelle le réapprovisionnement doit être proposé. C'est cette quantité qu'il s'agit de calculer à partir des données énumérées précédemment. Soient alors q_s la quantité de produit actuellement en stock, v_1, v_2, ..., v_{12} les ventes réalisées sur ce produit lors des 12 derniers mois, d_r le délai moyen de réapprovisionnement et q_d le niveau de déclenchement à calculer. Ce calcul s'organise selon les étapes suivantes :

1. récupérer les données q_s, d_r et v_1, v_2, ..., v_{12} ;
2. calculer la moyenne arithmétique m_v des ventes quotidiennes de ce produit ;
3. calculer le niveau de déclenchement q_d ;
4. comparer q_s à q_d :
   - $q_s \leq q_d$ => proposer de réapprovisionner ;
   - $q_s > q_d$ => ne rien faire.

Les étapes précédentes décrivent le procédé de calcul de q_d. Elles constituent un algorithme pour le calcul du niveau de déclenchement du réapprovisionnement. À partir des données du problème, ici q_s, d_r et v_1, v_2, ..., v_{12}, des règles opératoires permettent d'obtenir un résultat défini. Mais pour accéder à ce résultat, il faut passer par un ensemble d'étapes intermédiaires : récupérer les données, calculer une moyenne... de la même façon qu'il fallait préparer d'abord une crème Chantilly pour réaliser un Saint-Honoré. Ces étapes intermédiaires ne sont pas décrites : on ne dit pas comment calculer une moyenne, comment calculer le niveau de déclenchement q_d à l'aide de cette moyenne, comment préparer une crème Chantilly... Ces calculs intermédiaires, indispensables à l'obtention du résultat, ne sont pas explicités. L'apprenti cuisinier sait préparer une crème Chantilly, car c'est un préalable à la réalisation d'un Saint-Honoré ; alors, il dispose de tous les éléments pour réaliser la recette complète. L'informaticien, ou le chef magasinier, sait calculer la moyenne nécessaire et q_d pour déterminer quand réapprovisionner. Dans le cas contraire, il faudra étudier l'algorithme de calcul de la moyenne. Le cuisinier se reportera à la page 267 du livre de cuisine qui donne la recette qu'il ne sait pas encore préparer. Dans tous les cas, le problème est résolu grâce à un sous-algorithme qui est une solution à un sous-problème du problème initial.

Exercice 1 : Étapes de tâches quotidiennes

Énumérer les étapes élémentaires ordonnées pour réaliser les tâches quotidiennes suivantes :

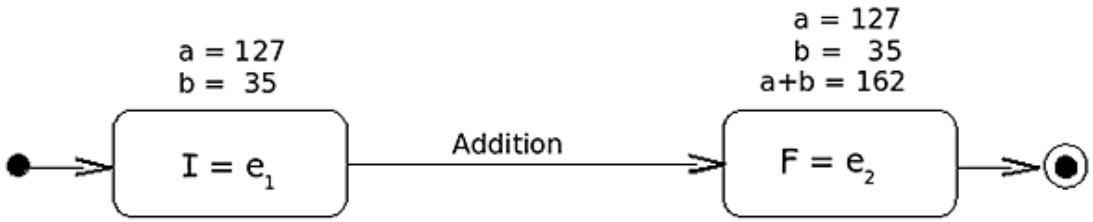
1. préparer du café ;

2. sortir la voiture du garage.

© ENI Editions - All rigths reserved

Définition informelle d'un algorithme

Un algorithme est la description d'un procédé. Il décrit une succession d'opérations élémentaires, exprimées dans un langage donné, à exécuter dans un certain ordre. L'exécution n'est possible que lorsque certaines conditions sont remplies. L'ail n'est ajouté que lorsqu'il a été haché ; on ne peut calculer le niveau de déclenchement q_d que lorsque la moyenne m_v des ventes quotidiennes a été obtenue. L'exécution des opérations élémentaires dont il s'agit permet de faire passer le système considéré, d'un état initial I, représenté par les valeurs des attributs donnés, à un état final F, représenté par les nouvelles valeurs des attributs et les résultats à obtenir. Pour l'addition, les attributs donnés sont les deux entiers a et b dont les valeurs sont respectivement 127 et 35. L'état initial est représenté par ces deux valeurs. Le résultat est 162 et l'état final est représenté par les trois valeurs 127, 35 et 162. La figure ci-dessous représente les deux états et la transition qui fait passer de l'un à l'autre.



L'état initial I ou e_1 est caractérisé par les valeurs 127 et 35 de a et b, respectivement. L'état final F ou e_2 est caractérisé par ces mêmes valeurs 127 et 35 et par la valeur 162 de la somme de a et b.

De même, l'état initial I de l'exemple de gestion de stock est constitué des attributs q_s, d_r et v_1, v_2, ..., v_{12}. L'état final F est représenté par les valeurs de q_s, v_1, v_2, ..., v_{12} et la valeur d'un attribut résultat, disons r, dont la valeur VRAI ou FAUX indique s'il y a lieu de réapprovisionner ou non.

Un algorithme correspond donc à un processus décomposé en étapes dont l'enchaînement permet d'atteindre un but fixé. D'où une définition informelle d'un algorithme :

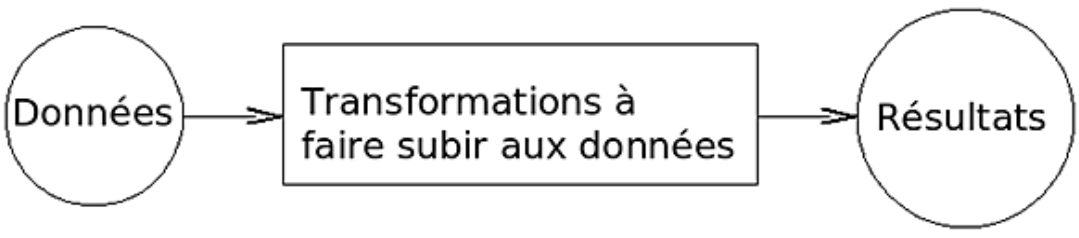
Définition

On appelle algorithme un ensemble de règles opératoires et de procédés définis en vue d'obtenir un résultat déterminé au moyen d'un nombre fini d'opérations séquentiellement ordonnées.

Les exemples des paragraphes précédents montrent que tout algorithme est caractérisé par les éléments suivants :

- les opérations à réaliser à chaque étape ;
- l'ordre de succession des différentes étapes ;
- l'existence de conditions déterminant ou non certaines étapes ;
- un début et une fin.

Les opérations dont il s'agit sont destinées à être exécutées par une machine fictive afin de réaliser les transformations des données qui produiront les résultats attendus. On peut schématiser ce point comme sur la figure suivante :



Comme nous avons commencé à le voir sur l'exemple de l'addition ci-dessus, les transformations subies par les données font « voyager » le système logiciel entre des états caractérisés par les valeurs de certaines données. Nous y reviendrons.

Cependant, il n'existe pas d'algorithme au sens propre quand on utilise un « langage » suffisamment expressif. Ainsi, considérons l'exemple suivant :

© ENI Editions - All rigths reserved

Exemple 5

On donne deux entiers a et b. Calculer leur somme.

C'est le problème de l'addition, déjà abordé ci-dessus, mais cette fois, supposons que nous disposions d'un langage qui « connaît » l'addition. Autrement dit, le simple fait de demander l'addition, en exhibant les nombres a et b à additionner, suffit à produire le résultat attendu. La situation est alors schématisée par le dessin de la figure ci-dessous, dans laquelle le bloc des transformations à faire subir aux données est vide.

Le « langage » utilisé est habituellement suffisamment expressif. Il suffit d'exprimer le résultat pour obtenir la solution. Ici, le résultat est la somme de deux entiers. Il n'y a pas d'algorithme à définir quand il est possible d'additionner et d'écrire directement :

```
Le Résultat est a+b
```

Pour la recette du Saint-Honoré, si la préparation de la crème Chantilly fait partie des compétences du cuisinier, il est en mesure de réaliser les objectifs de la recette et il est inutile de détailler les opérations nécessaires pour obtenir cette crème. De même lorsque le calcul de la moyenne arithmétique fait partie des opérations de base de l'environnement, il est possible de faire appel à ce calcul pour obtenir le niveau de déclenchement puis en déduire la décision à prendre en ce qui concerne le réapprovisionnement.

Cependant, imaginons la situation suivante, dans laquelle on ne dispose pas, dans l'état actuel de nos connaissances, de l'opération d'additions des entiers. On fait l'hypothèse que la seule opération connue est l'incrémentation, par laquelle on ne sait qu'ajouter 1 à un entier. Comment alors additionner deux entiers ? Cette fois, le cadre des transformations de la figure précédente n'est plus vide. Il devient nécessaire de préciser les étapes des transformations pour faire passer le système de l'état initial I, dans lequel on donne les valeurs de deux entiers a et b, à l'état final F dans lequel on obtient la somme a+b.

Les deux figures précédentes ne décrivent pas complètement la réalité de l'environnement dans lequel on étudie la résolution d'un problème par un algorithme. Les données sont habituellement accompagnées d'un *répertoire d'actions* conditionné par l'expressivité du langage utilisé pour étudier et exprimer l'algorithme. Dans l'exemple précédent, si ce répertoire contient l'addition des entiers, il n'est pas nécessaire d'étudier un algorithme pour résoudre le problème. Si, au contraire, l'addition n'est pas une opération de ce répertoire, l'étude algorithmique du problème devient nécessaire. C'était le cas, par exemple, pour la multiplication des entiers sur les premiers micro-processeurs. Le répertoire des instructions micro-programmées de ces composants ne disposait pas de la multiplication et celle-ci devait être étudiée et implémentée à partir, par exemple, de l'addition.

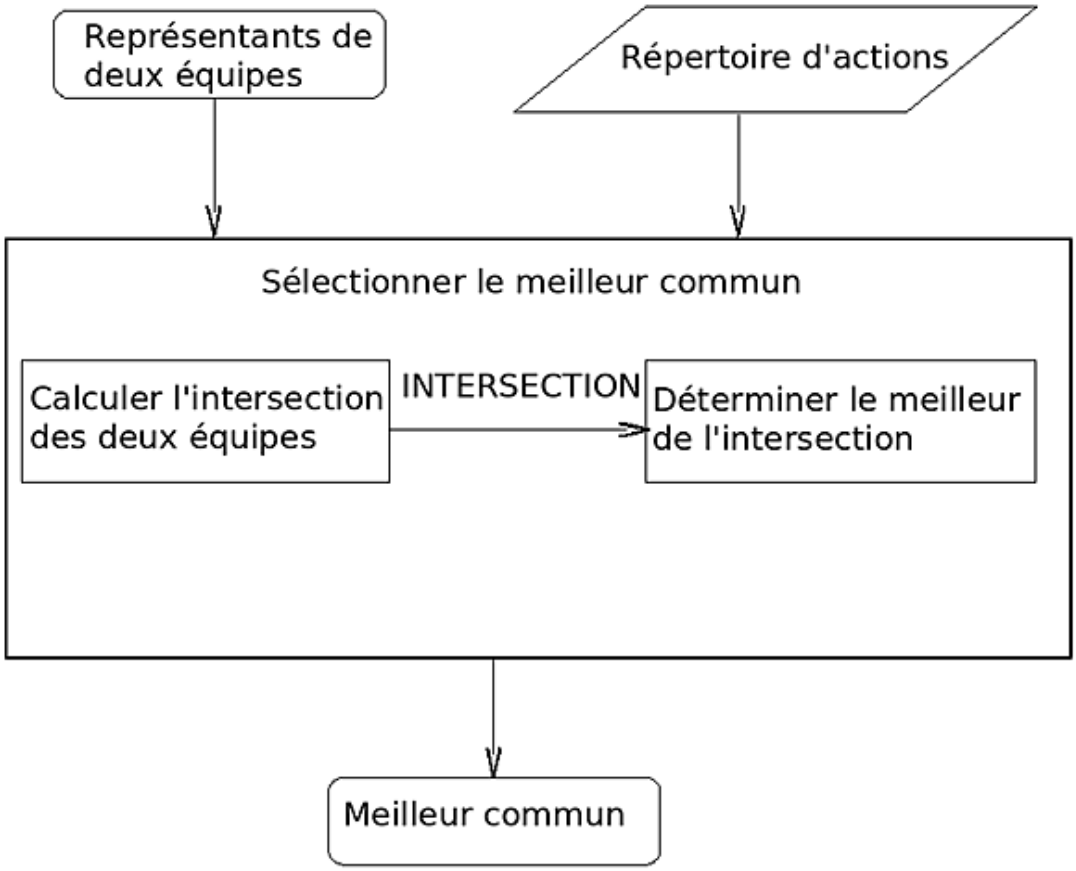
Ainsi, si le répertoire d'actions du langage utilisé dispose du calcul de la moyenne arithmétique, on l'utilise comme opérateur de base ; sinon, il devient nécessaire d'étudier le sous-algorithme qui calculera cette moyenne.

Exemple 6

Dans une entreprise, deux services emploient chacun une équipe de représentants. Certains représentants appartiennent aux deux services, mais pas tous. On veut attribuer une prime au seul représentant appartenant aux deux équipes et ayant réalisé le meilleur chiffre d'affaires sur l'année. Le problème consiste à déterminer le représentant qui sera primé.

Il est possible de représenter la situation comme sur la figure ci-dessous.

© ENI Editions - All rigths reserved



L'algorithme consiste donc à calculer l'intersection des deux ensembles de représentants, puis à déterminer, dans l'ensemble résultat, celui des représentants qui a le meilleur chiffre d'affaires sur l'année. Un langage qui dispose de l'opération d'intersection de deux ensembles de données, comme SQL par exemple, ne nécessite pas d'autre préalable à ce calcul et permet d'obtenir immédiatement le meilleur commun aux deux ensembles ; sinon, il faudra étudier un algorithme du calcul de cette intersection.

© ENI Editions - All rigths reserved

Spécifications

Une tâche importante, quand on cherche à écrire un algorithme, consiste à le *spécifier*. Il s'agit de dire, de la façon la plus précise possible, *ce que fait l'algorithme* et, cela, sans exprimer comment il le fait. En ce sens, spécifier un algorithme, c'est poser formellement le problème qu'il résout, comme l'a montré le chapitre précédent. C'est une étape difficile, mais indispensable pour clairement définir l'algorithme et ses effets. Cette section introduit, par quelques exemples et exercices résolus, cette étape de spécification. Elle est complétée dans la suite du livre, à mesure que nous construirons les outils nécessaires.

Exercice résolu 1 : Feux de circulation

On considère un feu qui règle la circulation à un carrefour.

Écrire l'algorithme qui détermine la couleur du feu lors du prochain changement.

Solution

Un tel feu signale l'état de la circulation qu'il règle par trois couleurs : vert, orange et rouge. Les couleurs s'enchaînent dans cet ordre à une cadence fixée. Lorsque la couleur est rouge, le changement refait passer le feu au vert. On ne s'intéresse qu'au changement de couleur, autrement dit, à l'opération qui fait passer de l'une à l'autre, dans l'ordre précisé.

Supposons que nous disposons d'une opération, désignée par **successeur**, qui prend en entrée une couleur de l'ensemble C = {rouge ; vert ; orange} et qui retourne son successeur dans la suite ordonnée (vert ; orange ; rouge). Avec ces hypothèses, l'algorithme s'écrit :

Algorithme 1 : Changement de la couleur d'un feu à un carrefour : version 0.1

```
algorithme : changer la couleur
Entrée
    c : une COULEUR de l'ensemble {vert ; orange ; rouge}
Effet
    c = successeur de ancienne valeur de c
```

Cette notation précise le nom de l'algorithme : **changer la couleur**. Elle indique aussi que cet algorithme utilise une donnée, dont le type est **COULEUR** et désignée par la lettre c, pour réaliser son calcul. C'est l'objet du paragraphe qui débute avec le mot **Entrée** que de donner cette précision. Le domaine des valeurs de c est clairement précisé : c'est l'ensemble {vert ; orange ; rouge}. Ce qui précède détaille les pré-requis pour le fonctionnement de l'algorithme.

Lorsque l'algorithme se termine et produit les résultats de son travail, on obtient un nouvel état du feu de circulation. Ce nouvel état est entièrement décrit, au moins pour cet exercice simplifié, par la couleur du feu ; c'est ce que dit le paragraphe **Effet**. On y lit que la nouvelle valeur de c, c'est-à-dire la nouvelle couleur du feu, est la valeur qui succède à l'ancienne valeur de cette couleur. La construction **ancienne** valeur de c est mise pour indiquer que la valeur de c avant modification par l'algorithme est utilisée. On voit donc que deux valeurs distinctes de c sont concernées : celle avant l'exécution de l'algorithme, représentée par la construction **ancienne** valeur de c, et la nouvelle valeur, celle calculée par l'algorithme, désignée par c. Il convient de noter que l'ancienne valeur de c disparaît lorsque l'algorithme se termine. Ainsi, cette valeur est modifiée par l'algorithme. C'est la responsabilité de **changer la couleur** de modifier cette valeur. Comme la valeur de c capture l'état du système logiciel, la modification de cette valeur modifie cet état.

Pour spécifier l'effet de l'algorithme, on utilise la construction c = **successeur** de **ancienne** valeur de c. Il faut encore dire ce que fait **successeur**, autrement dit, le résultat que produit cet opérateur. Ce sera fait plus bas. Pour l'instant, restons-en à l'idée intuitive sous-jacente.

Cette façon d'écrire l'algorithme n'est pas obligatoire. Ainsi, par exemple, on aurait pu utiliser d'autres notations pour exprimer la même idée. En voici une autre :

Algorithme 2 : Changement de la couleur d'un feu à un carrefour : version 0.2

```
changer la couleur
Donnée
    c : une COULEUR de l'ensemble {vert ; orange ; rouge}
Effet
    (ancien c = vert   => c = orange)
  ou sinon
    (ancien c = orange => c = rouge)
  ou sinon
```

© ENI Editions - All rigths reserved

```
    (ancien c = rouge  => c = vert)
```

Les expressions utilisées expriment la même condition que précédemment : la couleur orange succède au vert, rouge succède à orange et vert succède à rouge. Accessoirement, il n'est plus indiqué sur la première ligne qu'il s'agit d'un algorithme : c'est évident. **Entrée** a été remplacé par **Donnée**. Toute autre méthode pour préciser les données sur lesquelles travaille l'algorithme aurait été légitime.

La différence essentielle qui distingue cette description de la précédente est la façon dont elle exprime l'effet de l'algorithme. Au lieu de faire appel à un sous-algorithme **successeur** qu'il faudra plus tard définir, ici l'algorithme précise, pour chacune des valeurs possibles de la donnée d'entrée, la valeur que prendra c en sortie. C'est une expression qui est une disjonction de clauses booléennes. La notation :

```
ancien c = vert => c = orange
```

exprime que, lorsque la valeur de c en entrée est vert, elle est orange en sortie. Autrement dit, le calcul de l'algorithme consiste à remplacer la couleur vert par la couleur orange. Le symbole => est le symbole de l'implication en Mathématique.

Précisons davantage. Au lieu d'utiliser des nombres, nous utilisons des données d'un type particulier : des **COULEURs** dont les valeurs sont assujetties à évoluer dans un ensemble donné. Cet ensemble est défini ici par énumération. Il est possible aussi de donner une définition de ce type, pour préciser entièrement les données qui en relèvent.

```
type
    COULEUR
contrainte de domaine
    {vert ; orange ; rouge}
fin COULEUR
```

L'expression « contrainte de domaine » exprime que toute donnée de type **COULEUR** est assujettie à prendre sa valeur dans l'ensemble précisé. Là encore, ce n'est pas la seule façon de définir ce type. En voici une autre :

```
type COULEUR
    défini par l'énumération {vert ; orange ; rouge}
fin  COULEUR
```

Ayant posé cette définition, on considère que nous disposons d'un type **COULEUR** comme il existe des nombres entiers ou des nombres décimaux. On peut alors simplifier l'expression de l'algorithme 1 avec les conventions suivantes. Le nom de l'algorithme est fait d'une chaîne de caractères unique : **changer_couleur**, ou **changerCouleur**, ou ce que l'on voudra comme convention, pourvu qu'elle soit claire, sans ambiguïté. On précise ensuite le type des données sur lesquelles il travaille en entrée, c'est-à-dire les données qu'il doit recevoir et sur lesquelles il opère pour réaliser l'objectif qui lui est assigné : **Entrée** c : **COULEUR**. On indique également que cet algorithme a pour objectif de modifier la donnée qu'il reçoit en entrée et que cette modification est sa raison d'être, sa responsabilité. Il s'agit donc de préciser clairement que la donnée c est à la fois une donnée d'entrée, à partir de laquelle l'algorithme va établir son calcul, mais aussi une donnée de sortie, dont la nouvelle valeur est le but du calcul effectué. Il existe différentes conventions qui peuvent être adoptées pour préciser simplement ces points. En voici une :

```
Entrée Sortie c : COULEUR
```

La convention qui sera utilisée dans ce livre est une mention par défaut. Lorsque l'algorithme modifie certaines données qu'il reçoit en entrée, on ne précise rien de particulier. Ainsi donc, c'est lorsqu'il n'en modifie aucune qu'une mention explicite sera utilisée. Ce sera fait plus bas. Pour ce qui concerne l'algorithme particulier de changement de couleur, le lecteur sait qu'il modifie la donnée en entrée parce que rien ne vient infirmer ce fait. Cependant, un autre signe indique cette particularité.

L'algorithme retient la valeur de la donnée d'entrée. Il restitue cette valeur lorsqu'il s'agit de préciser l'effet de l'algorithme. Pour obtenir cette valeur, la construction **ancienne** valeur de c ou **ancien** c a été utilisée précédemment. Cette construction signifie que l'algorithme utilise un opérateur, noté ici **ancien** par exemple, et que cet opérateur permet de déterminer la valeur de c avant l'exécution du calcul. Ainsi donc, d'une valeur c donnée, cet opérateur en détermine une autre. Pour cette raison, la notation fonctionnelle sera utilisée dans la suite et justifiée plus tard. Par conséquent, la valeur de c avant l'exécution de l'algorithme sera obtenue par **ancien**(c).

Il reste à dire clairement ce que fait l'algorithme. C'est le paragraphe introduit par **Effet** qui l'exprime. Au lieu du mot effet, c'est le mot **postcondition** qui sera utilisé :

```
postcondition
    c = successeur(ancien(c))
```

Cette clause exprime que la nouvelle valeur de c calculée par l'algorithme est égale au successeur de la valeur que possédait c avant l'exécution de l'algorithme. On ne dit pas comment ce résultat est calculé, mais seulement ce que fait l'algorithme : il calcule le successeur de c et l'utilise pour modifier c. Ainsi, l'expression désignée par **postcondition** est une expression qui est soit VRAIE si **changer_couleur** fait correctement son travail, soit FAUSSE sinon. C'est donc une

© ENI Editions - All rigths reserved

expression dont le résultat est **BOOLÉEN**. On parle dans ce cas de prédicat.

On peut donc à présent donner une définition de **changer_couleur** utilisant ces conventions :

Algorithme 3 : Changement de la couleur d'un feu à un carrefour - version 1.0

```
algorithme : changer_couleur
Entrée
    c : COULEUR
postcondition
    c = successeur ( ancien ( c ) )
fin changer_couleur
```

Lorsque l'opérateur **successeur** fait partie du répertoire de base du langage et que le type **COULEUR** a été défini, cet algorithme est entièrement spécifié. Cependant, le type **COULEUR** ayant été construit pour les besoins du problème particulier à résoudre, l'opérateur **successeur** n'est pas encore défini. Par conséquent, l'algorithme **changer_couleur** ne prendra tout son sens que lorsque l'opérateur sera complètement spécifié. Ce sera fait plus bas.

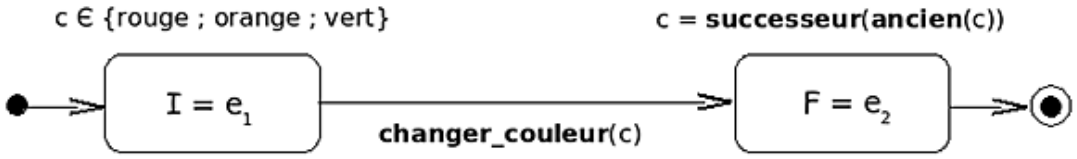
Cette définition de **changer_couleur** peut encore s'écrire de plusieurs façons équivalentes, utilisant des conventions légèrement différentes. En voici d'autres.

Algorithme 4 : Changement de la couleur d'un feu à un carrefour - version 1.0

```
changer_couleur(c : COULEUR)
postcondition
    c = successeur ( ancien ( c ) )
fin changer_couleur
```

Cette fois, les données d'entrée, celles à partir desquelles l'algorithme réalise son calcul, sont précisées derrière le nom de l'algorithme. C'est cette notation qu'utilisent tous les langages impératifs, avec parfois quelques variations syntaxiques dans la position des éléments. La première ligne est alors appelée la *signature* de l'algorithme.

Revenons au fonctionnement. **changer_couleur** fait passer le système d'un état initial I dans lequel le feu a une couleur c quelconque du domaine du type **COULEUR**, à un état final F dans lequel la couleur c du feu est devenue le successeur de l'ancienne couleur. La figure ci-dessous représente l'évolution de la situation par un diagramme d'états.



La transition de l'état initial à l'état final est assurée par l'opération **changer_couleur**. Finalement, le calcul du changement de couleur peut s'écrire comme ci-dessous :

Algorithme 5 : Changement de la couleur d'un feu à un carrefour - version 2.0

```
changer_couleur(c : COULEUR)
    # Modifier la couleur du feu qui règle la circulation d'un
    # carrefour.
précondition
    aucune
postcondition
    c = successeur ( ancien ( c ) )
fin changer_couleur
```

Les lignes :

```
# Modifier la couleur du feu qui règle la circulation d'un
# carrefour.
```

forment un commentaire. Il n'intervient pas dans la logique des opérations de l'algorithme. Son but est de clarifier une situation en apportant une précision supplémentaire par rapport aux instructions opérationnelles. La clause introduite par **précondition** indique les conditions que doivent satisfaire les données pour que l'algorithme fasse correctement son

© ENI Editions - All rigths reserved

travail. Ici, aucune condition particulière n'est imposée aux données d'entrée.

Il est important de bien se convaincre que cet algorithme ne dit rien sur la façon dont il calcule la couleur que doit prendre le feu. Il ne dit pas *comment* il réalise ce calcul. Il ne dit que ce qu'il fait : calculer la prochaine couleur du feu en utilisant le successeur de la couleur actuelle c qu'il a reçue en entrée. C'est la nouvelle couleur, qui remplace l'ancienne dans c, qui résulte du calcul de **changer_couleur**.

Pour que cette définition soit complète, il s'agit, à présent, de préciser ce que fait l'opération **successeur**. Cette précision n'est nécessaire que s'il n'est pas un opérateur du répertoire d'instructions de notre « langage ». Ainsi, si une **COULEUR** était un nombre entier par exemple, il ne serait peut-être pas nécessaire de définir cet opérateur.

Pour le définir, il suffit de préciser sa signature et la valeur du résultat qu'il calcule ; c'est ce que fait l'algorithme ci-dessous.

Algorithme 6 : Opérateur successeur - version 1.0

```
Algorithme successeur
    # Le successeur de c dans la liste (rouge ; vert ; orange).
Entrée
    c : COULEUR
Résultat : COULEUR
précondition
    aucune
postcondition
    # c n'est pas modifié.
    ancien(c) = c
    # Définition du résultat calculé.
      (c = vert   => Résultat = orange)
    ou sinon
      (c = orange => Résultat = rouge)
    ou sinon
      (c = rouge  => Résultat = vert)
fin successeur
```

La postcondition est constituée de deux clauses. La première précise que la couleur c reçue en entrée par l'algorithme successeur n'est pas modifiée. La seconde exprime ce que fait l'algorithme.

Il existe une différence essentielle, *fondamentale*, entre cette opération et l'algorithme précédent. L'algorithme **changer_couleur** a pour but de modifier l'état du système. Il prend une couleur, valeur de c, et il la modifie. La valeur de c n'est plus la même que celle qu'elle avait en entrée lorsque l'algorithme se termine. Au contraire, **successeur** ne modifie rien ; c'est ce qu'exprime la première clause de la postcondition :

```
ancien(c) = c
```

La valeur de c à la fin de l'exécution de l'algorithme est la même que celle de c lorsque l'algorithme commence son calcul. Il ne fait que calculer une nouvelle valeur à partir d'une donnée c qui reste ce qu'elle est, immuable. L'opérateur **successeur** est de la même nature qu'une opération, comme 2+3 par exemple. Cette opération prend deux nombres en entrée, 2 et 3 et calcule un résultat, 5, sans modifier les nombres 2 et 3. C'est ce que fait l'opération **successeur** : elle prend une couleur c et calcule un résultat, le successeur de c, sans modifier c. C'est aussi ce qu'exprime la notation :

```
Résultat : COULEUR
```

Elle signifie que **successeur** retourne un résultat de type **COULEUR**, comme 2+3 retourne un résultat de type **ENTIER**, sans modifier l'opérande, c'est-à-dire la donnée d'entrée c, elle-même de type **COULEUR**. L'effet de l'algorithme, précisé par les clauses de la postcondition, utilise encore le terme **Résultat**, qui exprime bien qu'une nouvelle valeur vient d'être obtenue.

La notation fonctionnelle introduite plus haut montre mieux encore cette différence de nature entre **changer_couleur** et **successeur** :

Algorithme 7 : Changement de la couleur d'un feu à un carrefour - version 2.1

```
successeur(c : COULEUR) : COULEUR
    # Le successeur de c dans la liste (rouge ; vert ; orange).
précondition
    aucune
postcondition
    # c n'est pas modifié.
```


© ENI Editions - All rigths reserved

```
    ancien(c) = c
    # Définition du résultat calculé.
      (c = vert   => Résultat = orange)
    ou sinon
      (c = orange => Résultat = rouge)
    ou sinon
      (c = rouge  => Résultat = vert)
fin successeur
```

Cette fois, les données d'entrées sont précisées dans la signature. La mention complémentaire indique que l'algorithme calcule un résultat et que ce résultat est de type **COULEUR**. Ainsi, trois éléments de la notation de ce type d'algorithme concourent à indiquer qu'il ne modifie pas les données qu'il reçoit en entrée pour effectuer ses calculs :

- la signature porte mention du type du résultat calculé et « retourné » à l'utilisateur logiciel de ses services ;
- une pseudo-variable notée **Résultat** reçoit la valeur calculée par l'algorithme. C'est cette valeur qui est « retournée » au client logiciel lorsque l'algorithme se termine. Ainsi, la valeur rendue au client est celle que possède cette pseudo-variable lorsque l'instruction **fin** est atteinte ;
- la postcondition contient une clause qui indique que la valeur de chacune des données d'entrée reste constante pendant les calculs de l'algorithme.

Le type du résultat, indiqué dans la signature de l'algorithme, est le type de la pseudo-variable **Résultat** qui n'est donc pas déclarée en tant que telle.

On peut remarquer, cependant, que la conjonction de ces trois conventions est redondante. En effet, les deux premières suffisent à qualifier l'algorithme pour lequel on peut accepter implicitement qu'il ne modifie pas les données en entrée, dès lors que la pseudo-variable **Résultat** apparaît. Le simple fait de donner la signature de l'algorithme montre qu'il est de la classe des opérations qui calculent un résultat sans modifier les données sur lesquelles il opère, comme une addition ne modifie pas les opérandes pour calculer leur somme. Ainsi, la mention `ancien(c) = c` n'est pas nécessaire et, pour l'algorithme de l'opération **successeur**, on peut affirmer que c'est un prédicat implicite qui participe à sa postcondition. On s'autorisera donc à ne pas le préciser lorsqu'aucune ambiguïté n'est à craindre.

Enfin, ici encore, on ne dit que ce que fait l'algorithme, sans dire comment il le fait. C'est que dire comment il fait son travail est plus difficile. Ce sera étudié plus tard.

Finalement, nous pouvons résumer ce que nous avons appris dans cet exemple.

Un algorithme est écrit sans contrainte de syntaxe autre que celle qui permet de l'exprimer sans ambiguïté. Ainsi, les conventions d'écriture exposées dans cette section ne sont pas obligatoires. Il est possible d'en adopter d'autres et le seul critère qui doit guider l'auteur de l'algorithme est sa correction et sa lisibilité.

Nous avons mis en évidence deux classes d'algorithmes. Les uns prennent des données en entrée pour calculer un nouveau résultat et ce calcul ne modifie pas les données utilisées. La signature d'un tel algorithme fait apparaître le type du résultat calculé. De plus, ce résultat est symbolisé par l'expression **Résultat** qui est donc la valeur produite par l'algorithme au bénéfice des logiciels clients de ses services. Ce résultat devient disponible dès que l'algorithme se termine, c'est-à-dire juste avant l'expression de la fin annoncée. Un tel algorithme sera appelé une *fonction* dans la suite du livre. Il ressemble à une fonction mathématique qui calcule un résultat à partir de paramètres sans les modifier. Il a donc le statut d'une *requête* qui « interroge » le système logiciel pour obtenir une information qui dépend de son état sans modifier cet état.

Définition

Une fonction (requête) est un algorithme qui rend un résultat sans modifier l'état du système logiciel.

L'autre classe d'algorithmes regroupe ceux qui ne calculent pas un résultat, mais qui modifient certaines données qui leur sont communiquées. Ils modifient donc l'état du système logiciel. Un tel algorithme ressemble à une *commande* envoyée au système logiciel pour lui demander de modifier son état actuel : c'est une *procédure*. La signature d'une procédure précise quelles sont les données en entrée et, parmi ces données, celles qui sont modifiées, c'est-à-dire celles destinées à recevoir de nouvelles valeurs réalisant ainsi un nouvel état du système. Lorsqu'une donnée en entrée voit sa valeur modifiée par l'algorithme, celui-ci « retient » la valeur initiale et cette valeur initiale peut être utilisée dans la postcondition à l'aide de la construction **ancien**.

Définition

Une procédure (commande) est un algorithme dont le calcul modifie l'état du système logiciel.

La suite de cette section étudie des exemples moins triviaux.

Exercice résolu 2 : Division euclidienne de deux entiers

© ENI Editions - All rigths reserved

On suppose que la division euclidienne de deux entiers n'est pas une opération du répertoire de notre langage d'implémentation.

1. Donner les spécifications de l'algorithme qui calcule le quotient entier de deux entiers positifs.

2. Donner les spécifications de l'algorithme qui calcule le reste de la division entière de deux entiers positifs.

Solution

Le texte de l'énoncé est clair : on demande les spécifications des algorithmes, autrement dit ce que font ces algorithmes et non pas comment ils font leurs calculs. Dans ce cas, le problème est simple.

Soient a et b deux entiers, positif ou nul pour a qui sera le dividende, strictement positif pour b qui sera le diviseur. Le quotient entier et le reste de la division euclidienne de a par b sont respectivement les entiers q et r tels que :

$a = b \times q + r$ avec $0 \leq r < b$

Le quotient est, par définition, l'unique entier q tel que :

$b \times q \leq a < b \times (q+1)$ (équation 1)

De plus, étant donnés a, b et q, on obtient r par :

$r = a - b \times q$ (équation 2)

On en déduit les spécifications demandées.

Algorithme 8 : Calcul du quotient euclidien de deux entiers

```
Algorithme quotient
    # Le quotient euclidien de a par b.
Entrée :
    a, b : ENTIER
Résultat : ENTIER
précondition
    a ≥ 0
    b > 0
postcondition
    b x Résultat ≤ a < b x (Résultat + 1)
fin quotient
```

L'algorithme calcule un résultat : le quotient entier de a par b. Il utilise pour cela des données, a et b de type **ENTIER**, sans les modifier : c'est une fonction. Le résultat calculé est un **ENTIER** : c'est dit dans la signature de l'algorithme. Cette signature aurait pu s'écrire aussi :

```
quotient(a, b : ENTIER) : ENTIER
    # Le quotient euclidien de a par b.
```

Cependant, la situation est nouvelle cette fois. Cet algorithme exige des conditions sur les données pour pouvoir assurer ses responsabilités. Il est clairement dit que a doit être positif ou nul et b strictement positif. La conjonction de ces deux contraintes imposées aux données en entrée forment la précondition. Ainsi, l'algorithme annonce qu'il ne fera correctement son travail qu'à ces conditions et qu'il est de la responsabilité du client logiciel de s'assurer qu'elles sont remplies avant de demander à l'algorithme de calculer le quotient. L'algorithme, quant à lui, assure que lorsque ces conditions sont satisfaites, il rend un **Résultat** vérifiant les assertions énoncées dans la postcondition.

La spécification de l'algorithme de calcul du reste consiste à utiliser la formule de calcul de l'équation (2) ci-dessus.

Algorithme 9 : Calcul du reste de la division euclidienne de deux entiers

```
Algorithme reste
    # Le reste de la division euclidienne de a par b.
Entrée :
    a, b : ENTIER
Résultat : ENTIER
précondition
```


© ENI Editions - All rigths reserved

```
    a ≥ b
    b > 0
postcondition
    Résultat = a - b x quotient(a, b)
fin reste
```

Remarquez que ce dernier algorithme utilise le résultat calculé par le précédent pour exprimer ce qu'il fait. Cela ne signifie pas qu'il utilise **quotient** pour calculer son propre résultat. En effet, cette spécification ne dit rien sur la façon dont **reste** calcule son résultat. Il dit seulement que son calcul consiste à établir un résultat égal à ce qu'annonce la postcondition. En ce sens, cette spécification ne fait que donner une définition du reste de la division.

Exercice résolu 3 : Calcul d'une moyenne arithmétique

Vous êtes professeur et vous voulez calculer la moyenne arithmétique des notes obtenues dans votre matière par Alain DUPONT.

1. Quels éléments sont nécessaires pour ce calcul ?

2. Comment spécifier ce calcul à l'aide des éléments recensés à la question précédente ?

Solution

La moyenne arithmétique est le résultat du quotient de la somme des notes par le nombre de notes. Les éléments nécessaires à ce calcul sont le nombre de notes et les notes proprement dites. Posons k > 0 le nombre de notes et n_1, n_2, ..., n_k les notes.

La spécification du calcul peut s'écrire alors :

```
Entrée
    n1, n2, ..., nk : RÉEL # Les notes dont on veut la moyenne.
    k : ENTIER             # Le nombre de notes.
Résultat : RÉEL            # La moyenne arithmétique des notes.
précondition
    # Une note est comprise entre 0 et 20.
    (∀i, 1 ≤ i ≤ k) (0 ≤ ni ≤ 20)
    # Le nombre de notes n'est pas nul.
    k ≥ 1
postcondition
```

$$\frac{\sum_{i=1}^{k} n_i}{k}$$

La spécification de ce problème met en œuvre avantageusement le symbolisme mathématique dont la précision et la concision sont incomparables. Cependant, cette notation n'est pas obligatoire et toute notation, qui définit clairement et sans ambiguïté le problème à résoudre, est acceptable. Ainsi, par exemple, l'expression de la postcondition peut être :

```
postcondition
    Le quotient par k de la somme des notes n1, n2, ..., nk
```

Exercice résolu 4 : Calcul des intérêts d'un compte d'épargne

Monsieur Jean-Paul DURAND possède un livret à la Caisse d'épargne. En fin d'année, il souhaite calculer lui-même le montant des intérêts rapportés en un an par son capital.

1. Quels sont les éléments nécessaires à ce calcul ?

2. Spécifier le calcul du montant des intérêts.

Solution

Le montant des intérêts annuels est obtenu en multipliant le montant du capital par le taux de placement et en divisant le produit par 100. Les données en entrée sont donc le montant du capital placé et le taux de placement.

```
Entrée
    capital : RÉEL # Montant du capital.
```

© ENI Editions - All rigths reserved

```
    taux    : RÉEL # Taux de placement.
Résultat : RÉEL
précondition
    capital ≥ 0
    taux > 0
postcondition
    Résultat = capital x taux x 0,01
```

La section suivante commence à étudier des algorithmes complets.

© ENI Editions - All rigths reserved

Premiers algorithmes

Cette section complète la précédente en montrant, sur quelques exemples élémentaires, les notations utilisées pour dire *comment* un algorithme effectue son calcul, autrement dit, comment assurer la transition de l'état initial I à l'état final F du système logiciel. On en reste encore, comme dans tout ce chapitre, à des idées intuitives, non formalisées, mais pourtant rigoureuses. Ici aussi, la méthode est exposée à partir d'exercices résolus.

Exercice résolu 5 : Réapprovisionnement de stock

Cet exercice reprend l'étude de l'exemple 4 commencée au début de la section Premiers exemples.

Écrire les instructions qui résolvent le problème posé.

Il s'agit donc d'étudier quelles sont les transitions qui feront passer le système logiciel de son état initial I, dans lequel sont connues les valeurs des données d'entrée du problème, à l'état final F dans lequel on sait s'il faut ou non lancer un réapprovisionnement. Nous avons vu, lors de l'étude préliminaire de cet exemple, que les étapes du calcul s'organisent selon le plan suivant :

- récupérer les données q_s, d_r et v_1, v_2, ..., v_{12} ;
- calculer la moyenne arithmétique m_v des ventes quotidiennes de ce produit ;
- calculer le niveau de déclenchement q_d ;
- comparer q_s à q_d :
  - $q_s \leq q_d$ => proposer de réapprovisionner ;
  - $q_s < q_d$ => ne rien faire.

Cet algorithme résout le problème et il est complet si on sait calculer une moyenne. Pour réussir à présenter la solution de problèmes plus complexes, pour lesquels cette forme simple d'écriture en langage naturel n'est pas adaptée, nous allons écrire la solution trouvée en utilisant un formalisme un peu plus restrictif que le langage naturel.

Il s'agit donc, à présent, de dire comment sont effectués les calculs de chaque étape pour prendre la décision.

On obtient la moyenne m_v des ventes quotidiennes en divisant le total des ventes sur l'année par le nombre de jours ouvrables. Si on admet 25 jours ouvrables par mois, on a :

$$m_v = \frac{\sum_{i=1}^{i=12} v_i}{12 \times 25} = \frac{v_1 + v_2 + \ldots + v_{12}}{300}$$

Le niveau de déclenchement q_d est le produit de cette moyenne par le délai d_r de réapprovisionnement : $q_d = m_v \times d_r$

Comme les v_i, d_r et q_s sont des données du problème, nous disposons de tout ce qui est nécessaire pour organiser le calcul complet. Une première version est donnée ci-dessous en deux parties. La première reprend les spécifications du problème. La seconde détaille les opérations à réaliser pour obtenir un résultat qui exprime s'il convient ou non de réapprovisionner.

Algorithme 9.1 : Spécifications du problème de réapprovisionnement

```
Algorithme déclenchement
    # Faut-il réapprovisionner ?
Entrée
    v1, v2, ..., v12 : ENTIER # Volumes des ventes mensuelles.
    dr : ENTIER # Délai de réapprovisionnement.
    qs : ENTIER # Seuil de déclenchement.
Résultat : BOOLÉEN
précondition
    toutes les données en entrée sont positives ou nulles.
```

© ENI Editions - All rigths reserved

Postcondition

Résultat = $(q_s \leqslant \frac{v_1+v_2+...+v_{12}}{300} \times d_r)$

fin déclenchement

La signature exprime que le calcul produit un résultat **BOOLÉEN**. Autrement dit, l'algorithme calcule un résultat égal à soit VRAI, soit FAUX, sans modifier les données utilisées pour réaliser ce calcul. Ces données sont des **ENTIERs** déjà présentés précédemment et elles ne sont soumises à aucune condition particulière, si ce n'est que l'algorithme attend en entrée des entiers non négatifs. Cette condition est exprimée ici en français, mais elle aurait pu tout aussi simplement utiliser les symboles mathématiques usuels dans ce cas.

La postcondition est un peu délicate et elle mérite quelques explications. Elle est exprimée sous la forme d'une égalité. À gauche, on trouve la pseudo-variable **Résultat** dont la valeur est celle rendue par l'algorithme lorsqu'il se termine. À droite du signe égal, on trouve une expression booléenne. Elle prend la valeur VRAI lorsque q_s est inférieur ou égal au membre de droite de la comparaison. Elle prend la valeur FAUX sinon. Dans tous les cas, c'est cette valeur, soit VRAI, soit FAUX, que doit prendre **Résultat** et que vérifie la postcondition. Il faut bien comprendre que le résultat de l'évaluation du membre de droite de l'égalité est une valeur dans l'ensemble {VRAI ; FAUX} et que c'est l'égalité de **Résultat** avec cette valeur qu'exprime la postcondition.

Passons à la deuxième partie de cette analyse. Il s'agit de dire comment passer de l'état dans lequel les données d'entrée vérifient la précondition à l'état qui vérifie la postcondition. C'est l'algorithme suivant qui exprime cette réalisation.

Algorithme 9.2 : Calcul de la décision de réapprovisionner (version provisoire)

```
Algorithme déclenchement
Réalisation
    # Calcul de la moyenne quotidienne des ventes.
    mv = (v1 + v2 + ... + v12) / 300
    # Calcul du niveau de déclenchement.
    qd <- mv x dr
    # Calcul de la décision.
    Résultat = (qs ≤ qd)
fin déclenchement
```

Le calcul de la moyenne m_v et du niveau de déclenchement q_d utilisent le symbole <- (flèche dirigée vers la gauche) pour exprimer que, par exemple, m_v prend la valeur calculée à droite de la flèche. Ce symbole est utilisé pour le distinguer du signe d'égalité qui exprime une condition booléenne dont la valeur est soit VRAI soit FAUX. Le signe <- est appelé *symbole d'affectation*. Il exprime que la variable placée à gauche prend la valeur placée à droite. Ainsi, m_v prend la valeur $(v_1+v_2+...+v_{12})/300$ et q_d prend la valeur $m_v \times d_r$.

Lorsque la quantité actuelle en stock q_s est inférieure ou égale au niveau de déclenchement, il faut émettre une proposition de réapprovisionnement. La responsabilité de l'algorithme consiste simplement à renvoyer VRAI comme résultat dans ce cas, ou FAUX dans le cas contraire. C'est ce que fait l'instruction annoncée par le commentaire *# Calcul de la décision*. La valeur à droite du symbole d'affectation est calculée. Elle prend la valeur VRAI lorsque le stock actuel est inférieur ou égal au niveau de déclenchement. C'est la valeur affectée à **Résultat** dans ce cas. Lorsque le stock actuel est strictement supérieur au niveau de déclenchement, $(q_s>q_d)$ est VRAI et par conséquent $(q_s \leq q_d)$ est FAUX. C'est alors cette valeur qui est affectée à **Résultat**. Ainsi, dans tous les cas, **Résultat** prend bien un valeur booléenne qui permettra de prendre la décision de réapprovisionner ou non.

Sous cette forme, l'algorithme n'est pas assez précis : il ne dit rien sur les symboles m_v et q_d. Il doit préciser que ce sont deux nombres réels, utilisés localement pour effectuer les calculs intermédiaires. Ces deux symboles n'ont d'existence que dans la fonction. On les déclare en donnant leur type, en début d'algorithme. On obtient alors la version plus détaillée ci-dessous.

Algorithme 9.3 : Calcul de la décision de réapprovisionner : version 1.0

```
Algorithme déclenchement
variables
    mv : RÉEL # Moyenne quotidienne des ventes.
    qd : RÉEL # Seuil de déclenchement du réapprovisionnement.
Réalisation
    # Calcul de la moyenne quotidienne des ventes.
    mv = (v1 + v2 + ... + v12) / 300
```


© ENI Editions - All rigths reserved

```
    # Calcul du niveau de déclenchement.
    q_d <- m_v x d_r
    # Calcul de la décision.
    Résultat = q_s ≤ q_d
fin déclenchement
```

La construction notée à l'aide de l'expression **variables** précise que ces symboles sont destinés à recevoir des valeurs qui participent au calcul du résultat. Il serait légitime d'insister sur le fait que ces symboles sont *locaux* à la fonction et qu'ils n'ont pas d'existence en dehors d'elle en écrivant : **variables locales**. Cependant, cela ne semble pas nécessaire dès lors qu'il est convenu que tout ce qui n'apparaît pas dans la signature est local. De même, il serait possible d'insister sur la nature de l'algorithme en précisant devant son nom **fonction** ou **procédure**, comme le font certains auteurs :

```
fonction déclenchement( liste des données en entrée) : BOOLÉEN
```

Là encore, cela ne semble pas nécessaire avec les conventions adoptées.

Exercice résolu 6 : Aire d'un disque

On considère un disque de rayon donné.

Écrire l'algorithme qui calcule son aire.

L'algorithme demandé calcule un résultat qu'il détermine à partir du rayon du disque, sans modifier ce rayon : c'est une fonction. Les Mathématiques indiquent, en appelant R le rayon et S l'aire du disque : S = 2 x π x R. Cette fois, le nombre π est une constante. En supposant cette constante connue, il est possible d'écrire :

Algorithme 10 : Aire du disque - version 1.0

```
Algorithme aire_du_disque
    # Aire du disque de rayon donné.
Entrée
    rayon : RÉEL # Le rayon du disque.
Résultat  : RÉEL
précondition
    rayon ≥ 0
réalisation
    Résultat <- 2 x π x rayon
postcondition
    Résultat = 2 x π x rayon
fin aire_du_disque
```

Cette solution calque directement la formule mathématique. Elle utilise le nombre PI dont la valeur est connue pour calculer un résultat élémentaire. Ce qui est nouveau ici, c'est l'expression de la postcondition comparée à celle de la réalisation du calcul. La réalisation du calcul exprime *comment* est obtenu le résultat. Ce n'est peut-être pas la seule façon de l'obtenir. La postcondition est une expression booléenne, dont la valeur est donc soit VRAI soit FAUX, qui exprime ce que fait l'algorithme : il rend un résultat égal à la valeur de l'expression à droite du signe d'égalité. La postcondition reste la même, même si la réalisation change pour une autre version du calcul.

Cette fonction est ainsi entièrement définie. Cependant, si on s'intéresse à son implémentation informatique, il faut choisir un langage de programmation et déterminer s'il propose une valeur pour la constante π. Selon le problème particulier qu'il s'agit de résoudre, il sera possible, par exemple, de choisir une valeur approchée pour cette constante ou d'utiliser une valeur plus précise si le langage la propose. En langage PHP par exemple, il est possible de définir la fonction ainsi :

```
<?php
define('PI', 3.141592) ;

function aire_du_disque($rayon)
    {
    assert('$rayon >= 0')          ; # Précondition.
    $s = ($rayon + $rayon) * PI    ; # Calcul de l'aire.
    assert('$s == 2 * $rayon * PI'); # Postcondition.
    return $s ;
    }
?>
```

Pour comprendre ces instructions, il faut peut-être quelques précisions sur la syntaxe du langage.

© ENI Editions - All rigths reserved

La clause **define** permet de définir une constante. La procédure **assert** provoque une erreur à l'exécution lorsque le prédicat qu'elle reçoit en entrée prend la valeur FAUX à l'exécution. Cette procédure permet d'implémenter d'une façon partielle et plutôt rustique la précondition et la postcondition afin qu'elles soient vérifiées à l'exécution.

En langage PHP, comme d'ailleurs en langage C ou en langage JAVA, le symbole de l'affectation est le signe d'égalité. Le signe d'égalité utilisé dans les expressions booléennes est le double signe == ou le triple signe ===. Par conséquent, x <- y s'écrit x = y en PHP. De même, l'égalité x = y se note x == y ou x === y en PHP. Comme les claviers d'ordinateurs ne permettent pas d'écrire certains symboles mathématiques, comme dans a ≥ b par exemple, on écrit a >= b. De même, a ≠ b s'écrit a != b en C et dans les langages dérivés.

Il est important de bien cerner les différences entre l'analyse algorithmique du problème et l'implémentation de la solution. L'analyse algorithmique exhibe la formule mathématique pour résoudre entièrement le problème. L'implémentation de la solution ajuste la programmation de cette formule sur deux points. Elle doit d'abord définir une valeur utilisable pour la constante. Elle remplace ensuite la multiplication du rayon par la constante 2 en lui substituant une addition. Cependant, ces variations n'affectent pas la spécification et la postcondition, en particulier, reste la même.

Cet exemple montre donc que la simple analyse algorithmique d'un problème ne suffit pas toujours à l'élaboration complète de la solution du problème informatique à résoudre. Il faut pourtant garder à l'esprit que les variations particulières imposées par les langages de programmation, notamment les variations syntaxiques, comme = pour l'affectation ou >= pour comparer deux valeurs, ne sont que des détails et qu'ils ne concernent pas l'algorithmique. Celle-ci n'impose aucune syntaxe et il semble préférable d'utiliser la syntaxe des Mathématiques qui sont un langage universellement accepté.

La section suivante développe la solution de quelques problèmes pour mettre en évidence certaines règles du domaine.

© ENI Editions - All rigths reserved

Exercices résolus

Cette section propose quelques exercices qui demandent un peu plus de réflexion. Il s'agit de préparer des solutions complètes et définitivement prêtes pour l'étape d'implémentation, même si un même problème peut donner lieu à différentes solutions toutes correctes. Ces propositions de solutions sont élaborées à la suite de chaque énoncé. Pour chacune d'elles, les notations sont complétées par les notions nécessaires à leur expression. Cependant, le lecteur est invité à réfléchir à ces problèmes, à les spécifier et, éventuellement, à les résoudre d'une façon informelle, par exemple en rédigeant les solutions « en français » avant d'étudier les solutions proposées.

Exercice résolu 7 : Échanger les valeurs de deux variables

*On donne deux variables quelconques, d'un type **T** qui n'est pas précisé.*

1. Écrire l'algorithme qui échange les valeurs de ces deux variables.

Soient V_1 la valeur de la donnée contenue dans la variable v_1 et V_2 la valeur dans v_2. Lorsque l'algorithme se termine, la donnée contenue dans la variable v_1 doit avoir la valeur V_2 et celle contenue dans v_2 doit avoir la valeur V_1.

Indication *: on donne une bouteille remplie d'eau et une autre bouteille remplie de vin. Comment échanger les contenus des deux bouteilles sans les mélanger ?*

L'indication qui précède suggère d'utiliser une variable intermédiaire pour résoudre le problème. Peut-on faire autrement ? C'est l'objet de la question suivante.

*On suppose que les objets de type **T** auxquels appartiennent les valeurs des variables v_1 et v_2 peuvent être additionnés ou soustraits. C'est le cas des nombres entiers ou réels par exemple.*

2. Même question que la précédente, mais l'algorithme proposé ne fait que des additions et des soustractions, sans utiliser d'autre variable que v_1 et v_2.

Solution

On peut commencer par spécifier l'algorithme. C'est ce qui est fait ci-dessous. Remarquez qu'il s'agit de modifier les données reçues en entrée. Les valeurs à échanger sont portées par des variables qui doivent être modifiées par l'algorithme : c'est donc une procédure.

Algorithme 11.1 : Spécifications de l'échange de deux variables

```
Algorithme échanger
    # Échanger les valeurs de a et b.
Entrée
    a, b : T # Variables dont les valeurs seront échangées.
précondition
    aucune
postcondition
    a = ancien(b)
    b = ancien(a)
fin échanger
```

Cet algorithme ne précise pas de résultat. C'est donc une procédure, qui modifie l'état du système logiciel. De quelle façon ? La postcondition l'exprime clairement : la valeur de `a` est celle de l'ancienne valeur de `b`, c'est-à-dire la valeur que possédait `b` avant que l'algorithme ne fasse ses transformations. La valeur de `b` est celle de l'ancienne valeur de `a`. Encore une fois, **ancien**(x) fait référence à la valeur de x avant le calcul de l'algorithme et non pas à la valeur acquise en cours de calcul.

Le type **T** n'est pas précisé. Par conséquent, les variables peuvent être d'un type quelconque. Ce type ne contraint pas la procédure qui « travaille » de la même façon, quel que soit ce type. Ceci termine la spécification et on peut envisager d'étudier comment l'algorithme peut faire ce travail.

L'énoncé suggère une solution qui indique clairement la méthode à utiliser : pour échanger les contenus de deux bouteilles, il suffit d'utiliser une bouteille supplémentaire dans laquelle on transvase l'un des deux liquides, disons le vin. Ainsi, la bouteille de vin devient libre pour recevoir l'eau. Cette opération libère la bouteille d'eau dans laquelle on peut à présent transvaser le vin contenu dans la bouteille supplémentaire.

Pour le problème informatique, on utilisera une variable supplémentaire qui recevra la valeur de l'une des deux variables à échanger, disons `a`. On peut alors copier la valeur que contient `b` dans `a` puis copier la valeur de la variable supplémentaire dans `b`. Soit `temp` cette nouvelle variable. Elle est destinée à recevoir une valeur de type **T** et doit donc être déclarée comme telle. On obtient la solution de l'algorithme ci-après.

© ENI Editions - All rigths reserved

Algorithme 11.2 : Échanger deux variables - version 1.0

```
Algorithme échanger
    # Échanger les valeurs de a et b.
Entrée
    a, b : T # Variables dont les valeurs seront échangées.
précondition
    aucune
variable
    temp : T
réalisation
    temp <- a    # Copier a dans temp. Commentaires à revoir.
       a <- b    # Copier b dans a.
       b <- temp # Copier temp dans b.
postcondition
    a = ancien(b)
    b = ancien(a)
fin échanger
```

Le cœur de l'algorithme est constitué du paragraphe introduit par **réalisation** qui réalise effectivement l'échange des valeurs. Ce paragraphe illustre une mauvaise façon d'utiliser les commentaires. Ici, les commentaires ne font que paraphraser les instructions de transformations des données. C'est une mauvaise pratique puisque ces commentaires n'apportent rien. Voyons comment justifier cet algorithme, ce qui suggérera comment le commenter. Dans la suite, on note par une lettre minuscule un « contenant », c'est-à-dire un nom de variable. Une lettre majuscule désigne un « contenu », c'est-à-dire la valeur de la variable.

Avant toute exécution d'instruction de réalisation, l'état du système est :

```
# État avant l'échange.
temp = ??? ; a = A ; b = B
```

L'instruction `temp <- a` copie la valeur de a dans temp. Le nouvel état est alors :

```
# État après copie de a dans temp : temp <- a.
temp = A ; a = A ; b = B
```

Il est important de remarquer que cette instruction réalise une copie de la valeur de a dans temp. Cette copie ne modifie pas la valeur `A` contenue dans `a`, à plus forte raison celle de `b`. Cependant la valeur inconnue dans temp, symbolisée par les ??? de l'état précédent, est définitivement perdue.

L'instruction `a <- b` conduit à un nouvel état :

```
# État après copie de b dans a : a <- b.
temp = A ; a = B ; b = B
```

Ainsi, la valeur contenue dans `a` est perdue, remplacée par `B` qui est copiée de `b`. Là encore, il s'agit d'une copie et donc `b` n'est pas modifiée. C'est ici que l'utilisation de temp apparaît plus clairement. Sans elle, la valeur `A` serait perdue.

Enfin, l'instruction `b <- temp` conduit à un nouvel état :

```
# État après copie de temp dans b : b <- temp.
temp = A ; a = B ; b = A
```

La valeur `A` contenue dans temp a remplacé la valeur `B` dans `b` par copie, sans modifier `temp`. Finalement, la situation obtenue laisse `B` dans `a` et `A` dans `b` et c'est ce qu'affirme la postcondition de **échanger**. Cette analyse montre que le paragraphe introduit par **réalisation** serait plus clair s'il était rédigé ainsi :

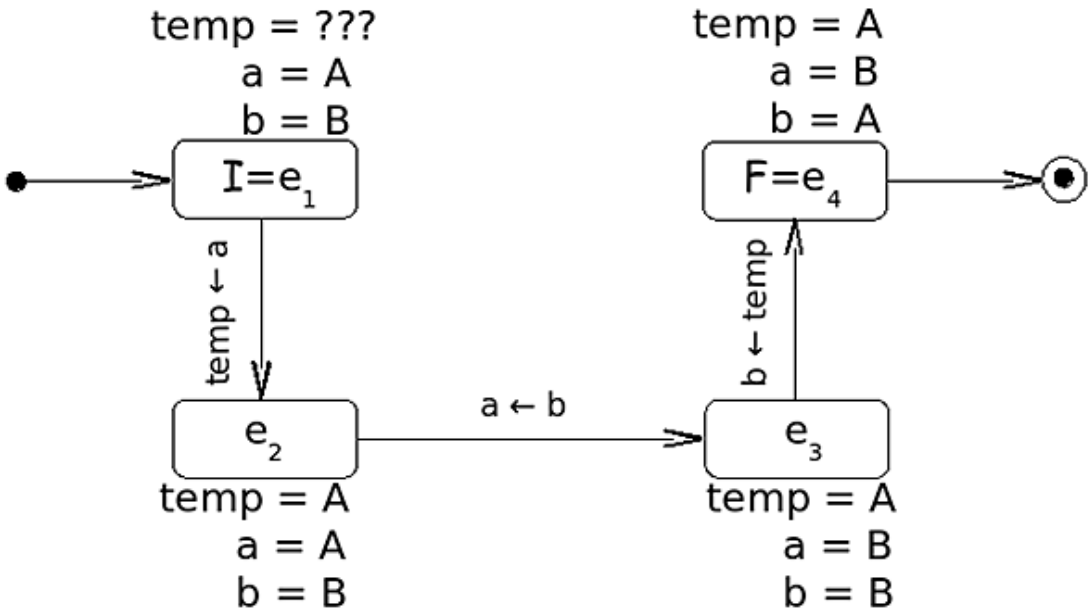
```
réalisation
    # État initial : a = A ; b = B.
    temp <- a    # temp = A ; a = A ; b = B.
       a <- b    # temp = A ; a = B ; b = B.
       b <- temp # temp = A ; a = B ; b = A.
    # État final :  a = B ; b = A.
```

Ce qui permet de se convaincre de la correction de cet algorithme, c'est le détail des situations obtenues après chaque instruction. C'est un point important, fondamental. Les instructions d'un algorithme disent « ce que fait » l'algorithme ou « comment » il le fait. Elles ne disent rien sur les situations obtenues c'est-à-dire sur l'état atteint après l'exécution de

© ENI Editions - All rigths reserved

l'instruction. Cet état est représenté par les valeurs des variables, leur contenu. C'est en détaillant ces situations, c'est-à-dire en analysant et en décrivant l'état du système, que l'on peut vérifier l'algorithme.

L'évolution des états peut aussi être représentée par le diagramme d'états de la figure ci-dessous.



Finalement, nous disposons ainsi d'une méthode d'analyse qui peut être utilisée pour résoudre la question suivante. Additionner `a` et `b` donne un résultat égal à `A+B`. En retranchant `B` à ce résultat on obtient `A` comme nouveau résultat :

```
a <- a + b => a = A + B et b = B
b <- a - b => b = A et a = A + B
```

Ainsi, `b` contient `A` et `a` contient `A+B`. En retranchant `A`, c'est-à-dire le contenu de `b` à `a`, on obtient `a = B`.

```
a <- a - b => a = B et b = A
```

Finalement, l'algorithme s'écrit :

Algorithme 11.3 : Échanger deux variables - version 2.0

```
Algorithme échanger
    # Échanger les valeurs de a et b.
Entrée
    a, b : T # Variables dont les valeurs seront échangées.
précondition
    (T,+) et (T,-) sont définies
réalisation
    a <- a + b    # a = A + B ; b = B.
    b <- a - b    # a = A + B ; b = A.
    a <- a - b    # a = B ; b = A.
postcondition
    a = ancien(b)
    b = ancien(a)
fin échanger
```

La précondition précise que **T** est un type de données sur lequel sont définies les deux opérations applicables + et -, d'addition et de soustraction.

Comme pour la version précédente, l'analyse de la solution consiste à suivre l'évolution de l'état du système. Le diagramme d'états peut encore être utilisé ici. Établir ce diagramme est laissé en exercice.

Exercice 2 : Diagramme d'états de la version 2 de échanger

Construire le diagramme d'états de cette dernière version de l'algorithme d'échange de deux variables.

© ENI Editions - All rigths reserved

Remarquez que les deux versions de l'algorithme sont radicalement différentes dans la façon qu'elles ont d'échanger les valeurs des deux variables. Les deux algorithmes sont pourtant identiques dans leurs spécifications lorsque **T** est un type numérique usuel par exemple. Ils font exactement « la même chose » : échanger les valeurs de deux données de même type **T**. Ils le font de deux façons différentes et qui n'ont pas les mêmes conséquences : le premier utilise une variable intermédiaire. Le second fait des calculs. On peut penser que le premier occupera plus de place dans la mémoire de l'ordinateur utilisé pour l'implémenter, alors que le second prendra plus de temps pour mettre le système dans l'état final. Cependant, un utilisateur qui ne serait pas concerné par la façon de réaliser les transformations, n'aura à connaître que les spécifications pour pouvoir mettre en œuvre **échanger**. Bien entendu, les deux algorithmes ne sont interchangeables que sur des données d'un type **T** sur lequel sont définies les opérations d'additions et de soustractions. C'est ce que précise la précondition de la seconde version. Par conséquent, seule la version 1 est générale et donc utilisable dans tous les cas.

À ce stade, ces deux algorithmes sont complets. Ils peuvent servir à coder une solution dans un langage de programmation d'ordinateur. Essayons en langage PHP par exemple.

Le script principal qui utilise la procédure **échanger** est, par exemple :

```
<?php
require_once('echanger.php') ; # Échange les valeurs de ses paramètres.
require_once('ecrire.php')   ; # Écrit les valeurs des paramètres.

$a = 17 ;
$b = 21 ;
ecrire('Principal avant échange', $a, $b)  ;
echanger($a, $b);
ecrire('Principal après échange', $a, $b)  ;

exit(0) ;
?>
```

Le fichier *echanger.php* est un lien sur le fichier *echanger1.php* qui contient la définition d'une première version de la procédure :

```
<?php
/**
 * Échanger les valeurs de "$a" et "$b".
 *
 * @param mixed $a
 * @param mixed $b
 */
function echanger($a, $b)
{
    $temp = $a ;
    $a    = $b ;
    $b    = $temp ;
}
?>
```

Remarquez que, pour PHP, tout est fonction. Cette procédure est donc déclarée à l'aide du mot **function**.

L'exécution du script principal qui appelle cette procédure et la procédure **ecrire** donne les résultats suivants :

```
Principal avant échange : A = 17  B = 21
Principal après échange : A = 17  B = 21
```

ce qui, manifestement, n'est pas ce qui est attendu. Pour comprendre ce qui se passe, ajoutons dans la procédure **echanger** les mêmes instructions de trace :

```
function echanger($a, $b)
{
    ecrire('Échanger ', $a, $b) ;
    $temp = $a ;
    $a    = $b ;
    $b    = $temp ;
    ecrire('Échanger ', $a, $b) ;
}
```

Cette fois, le résultat de l'exécution est :

© ENI Editions - All rigths reserved

```
Principal avant échange : A = 17      B = 21
Échanger  avant échange : A = 17      B = 21
Échanger  après échange : A = 21      B = 17
Principal après échange : A = 17      B = 21
```

Ainsi, on constate que la procédure d'échange s'acquitte correctement de sa tâche et réalise bien ce qui a été prouvé plus haut. Par contre, le script principal ne reporte pas les changements. Pour le script principal, les deux variables gardent les mêmes valeurs et les transformations réalisées par la procédure n'ont aucune influence sur les valeurs des variables dans le script principal.

La procédure **echanger** travaille sur des copies locales des paramètres $a et $b. Les noms des variables ne changent rien à l'affaire. Dans le script principal, l'utilisation de la procédure se fait par l'instruction **echanger**($a, $b) par laquelle on demande à échanger les valeurs des variables $a et $b. Ici, $a et $b sont les paramètres effectifs de l'appel. Dans la procédure, $a et $b sont des paramètres formels, c'est-à-dire des marque-places qui indiquent simplement que la procédure reçoit en entrée deux paramètres, le premier étant désigné par $a et le second par $b. Il aurait été légitime de les appeler $x et $y ou $premier et $second par exemple. De plus, la règle essentielle ici est que, partout où apparaît $a ou $b, c'est une *copie locale* qu'utilisera la procédure et non le paramètre effectif. Ainsi, à la sortie de la procédure, le contexte du script principal est restauré et $a et $b retrouvent les valeurs qu'elles avaient dans ce contexte.

Pour sortir de cette situation, il faut un moyen de forcer la procédure à utiliser les variables effectives et non pas des copies locales. C et les langages dérivés adoptent pour cela une convention syntaxique qui consiste, par exemple, à faire précéder d'un & chacun des paramètres formels de la procédure. La signature de la procédure devient ainsi :

```
function echanger(&$a, &$b) ;
```

Après cette modification d'une copie de la procédure dans un nouveau fichier *echanger2.php* et la redéfinition du lien *echanger.php* vers cette copie, on obtient à l'exécution du script principal :

```
Principal avant échange : A = 17      B = 21
Échanger  avant échange : A = 17      B = 21
Échanger  après échange : A = 21      B = 17
Principal après échange : A = 21      B = 17
```

ce qui est exactement ce que nous attendons.

Cette étude montre que, même après une étude soignée d'un problème, il est encore nécessaire de procéder à l'adaptation de l'algorithme aux particularités des langages de programmation. La convention mise en évidence pour le langage PHP ne lui est pas particulière. Tout module logiciel, procédure ou fonction, utilise des copies locales des valeurs des paramètres effectifs. C'est une convention syntaxique particulière qui contraindra le module à utiliser les valeurs des paramètres effectifs au lieu de copies locales. Bien entendu, l'algorithmique n'a pas à se soucier de ces règles de syntaxe. Les spécifications sont là pour définir précisément le contexte, les conditions d'usage et les effets du module et il n'y a pas lieu d'introduire des règles de grammaire supplémentaires pour préciser davantage.

Envisageons, à présent, l'implémentation dans un langage particulier de la seconde version de l'algorithme d'échange. La précondition impose aux valeurs à échanger d'être d'un type sur lequel est définie l'addition. Pour la suite de cette étude, supposons que ce type est le type **ENTIER** et que le langage de programmation est C. En langage C, comme dans tout langage de programmation d'ailleurs, le domaine des valeurs d'un entier est contraint par la représentation des nombres en machine. Ainsi, par exemple sur tel ordinateur courant, celui utilisé pour écrire ce texte, un entier est représenté sur 32 chiffres binaires (*bits*), c'est-à-dire sur 4 octets. Ainsi, sa valeur, lorsqu'il est positif, est limitée à INT_MAX = 2^{31}-1 soit à 2147483647. Tout dépassement de cette limite engendre une erreur de calcul et parfois même une erreur dès la phase de compilation du programme. Considérons alors notre version de **échanger**. La première instruction de la réalisation calcule la somme a+b et range le résultat dans a. Supposons alors des valeurs telles que a > INT_MAX / 2 et b > INT_MAX / 2. Leur somme est alors a+b > INT_MAX et le résultat ne peut être calculé. En toute rigueur, la précondition devrait donc imposer les contraintes associées sur le domaine des valeurs de a et b :

précondition

$a \leq INT_MAX$

$b \leq INT_MAX$

$a \leq INT_MAX - b$

Il n'est pas difficile de démontrer que cette précondition assure que la somme reste inférieure ou égale à INT_MAX. Il reste à compléter pour imposer à la somme de rester supérieure ou égale à INT_MIN de la même façon.

Ces considérations d'implémentation concernent-elles l'algorithmique ? Ce livre prétend que non. L'algorithme reste dans l'espace du problème. L'addition et la soustraction des **ENTIERs** sont bien définies, leurs propriétés sont connues et il ne semble pas nécessaire de s'attarder sur des particularités qui viennent nier ou amender ces propriétés. Bien entendu, programmer impose d'autres contraintes qui viennent moduler l'analyse préalable. Il faudra bien en tenir compte pour programmer, mais on sera alors dans l'espace de la solution et les contraintes de domaine ne seront pas les plus difficiles à satisfaire. Par conséquent, la suite de ce livre ne tiendra pas compte des contraintes imposées par les langages de programmation. Certaines difficultés seront signalées parfois lorsque la solution n'est pas triviale, mais nous en resterons habituellement à des problèmes simples.

© ENI Editions - All rigths reserved

Les rudiments étudiés dans les sections précédentes sont insuffisants pour étudier des problèmes plus complexes. Il nous faut encore préciser quelques règles fondamentales sur l'utilisation des mots du langage qui nous servent à repérer les données et les résultats.

Chaque donnée ou résultat est représenté par une *variable* dont on suppose qu'elle occupe un ensemble de positions dans la mémoire du calculateur utilisé pour coder la solution du problème. Appelons une « case » l'ensemble des positions occupées par une variable. Ainsi, la valeur de a occupe une case, la valeur placée dans temp en occupe une autre... L'utilisation de la mémoire d'une machine de VON NEUMANN est réglée par les trois axiomes suivants :

(A1) : on ne peut placer qu'une seule donnée à la fois dans une case ;

(A2) : on peut utiliser à tout moment le contenu d'une case sans le détruire ;

(A3) :on peut toujours remplacer le contenu d'une case par une nouvelle donnée du même type.

L'axiome (A3) sera dorénavant un axiome de la « théorie » de ce livre, mais n'est pas un axiome de tous les langages de programmation. Ainsi, par exemple pour un langage comme PHP, il est possible d'écrire $a = 55 ; puis, plus tard, $a = 'Bonjour' ; ce qui fait que l'entier 55 a été remplacé par une chaîne de caractères, qui n'est trivialement pas du même type. Cet axiome dit aussi qu'une donnée placée dans une case n'est jamais définitivement présente et qu'il est toujours possible de la remplacer par une autre valeur.

L'axiome (A2) est l'*axiome de copie*. Il dit que la copie du contenu d'une case vers une autre case, lors d'une affectation par exemple, ne détruit pas le contenu de la case source.

Mais qu'est-ce qu'une *variable* précisément ?

Définition

On appelle variable une instance d'un type de données.

Un *type* précise le *domaine des valeurs* de la variable, c'est-à-dire l'ensemble des valeurs possibles pour cette variable, et les *opérations applicables* à des données de ce type. Ainsi, par exemple, le type **ENTIER** définit les données de l'ensemble des entiers sur lesquels on peut effectuer les opérations usuelles d'addition, multiplication... Le type **COULEUR** définit les valeurs possibles des instances de couleurs : c'était rouge, orange et vert. La seule opération actuellement définie sur les données de ce type est **successeur**. Définir une variable c'est, en algorithmique, lui donner un nom et dire à quel type elle appartient, c'est-à-dire de quel type elle est une instance. Cette seule mention suffit pour préciser les valeurs qu'elle peut prendre et les opérations qui lui sont applicables. On peut toujours définir un nouveau type de données. Cette définition devra clairement préciser les valeurs possibles de ses instances et les opérations applicables.

L'exercice suivant illustre la création et l'utilisation d'un nouveau type de données. Il est instructif dans la façon de réutiliser des algorithmes pour en définir d'autres.

Exercice résolu 12 : Réservoirs

Un réservoir est un conteneur de liquide. On peut réaliser différentes opérations sur un réservoir, comme le remplir, le vider, ajouter du liquide à son contenu...

1. Établir la liste des opérations pertinentes applicables à un réservoir. Distinguer clairement les fonctions des procédures.

2. Écrire les spécifications puis les définitions des algorithmes de chacune de ces opérations.

Solution

Les opérations pertinentes sur un réservoir, comme d'ailleurs sur tout objet informatique, se répartissent en deux familles, comme nous avons commencé à le voir précédemment.

- Les *requêtes* permettent d'interroger l'objet sur son état : ce réservoir est-il vide ? Est-il plein ? Quelle est sa capacité ?... ;

- les *commandes* permettent de modifier cet état : remplir ce réservoir, le vider, lui ajouter une quantité donnée de liquide...

Une requête est implémentée par une fonction : elle calcule un résultat qu'elle retourne au client du service qu'elle représente. Une commande est réalisée par une procédure qui va modifier les valeurs de certaines variables d'état. Détaillons les membres de chaque famille pour un réservoir.

Les requêtes sur un réservoir sont :

- **estVide** qui détermine si un réservoir est vide ;

© ENI Editions - All rigths reserved

- **estPlein** qui détermine s'il est plein.

Les commandes sont :

- **vider** pour ramener sa contenance à 0 ;
- **remplir** pour ajuster sa contenance à sa capacité ;
- **ajouter** une quantité donnée de liquide pour augmenter sa contenance actuelle ;
- **enlever** une quantité donnée de liquide.

L'état d'un réservoir est entièrement caractérisé par deux nombres. Sa capacité donne en litres la quantité de liquide qu'il peut contenir. C'est un nombre décimal positif, éventuellement nul pour un réservoir qui ne peut rien contenir. Sa contenance donne en litres la quantité actuelle de liquide que contient le réservoir. C'est aussi un nombre décimal positif, éventuellement nul si le réservoir ne contient rien. Par conséquent, un **RÉSERVOIR** informatique est une structure de données qui rassemble indissociablement ces deux nombres pour en faire un réservoir. Elle est définie ainsi :

```
type
    RÉSERVOIR
structure
    capacité   : RÉEL
    contenance : RÉEL
fin RÉSERVOIR
```

Lorsque nous aurons besoin de définir une variable de type **RÉSERVOIR**, nous écrirons :

```
# Déclaration d'un variable de type RÉSERVOIR.
r : RÉSERVOIR
```

ce qui aura le même effet qu'une déclaration usuelle pour un nombre entier ou un nombre réel. Il devient alors possible d'initialiser la capacité et la contenance du réservoir r :

```
r.capacité <- 1000     # Initialise la capacité de r à 1000 litres.
r.contenance <- 0      # Sa contenance est nulle (réservoir vide).
```

Pour « interroger » le réservoir r ainsi défini et obtenir sa capacité par exemple, on écrira :

```
variable
    c : RÉEL
...
# Copier la valeur de la capacité du réservoir r dans c.
c <- r.capacité
```

Ainsi, étant donnée une variable r de type **RÉSERVOIR**, on accède à l'une de ses caractéristiques, sa capacité ou sa contenance, en utilisant la notation pointée :

`r.capacité` ou `r.contenance`.

Accessoirement, ces conventions montrent que deux nouvelles requêtes permettent d'interroger un réservoir sur son état :

- `c <- r.contenance` copie dans c la contenance du réservoir r ;
- `c <- r.capacité` copie de même sa capacité.

Nous disposons à présent de tout ce qui est nécessaire pour définir les opérations sur un réservoir. Commençons par les requêtes.

La fonction **estVide** interroge un réservoir r qu'elle reçoit en entrée. Elle rend le résultat VRAI si et seulement si le réservoir est vide, autrement dit, si et seulement si sa contenance est nulle. L'algorithme ci-dessous définit cette fonction.

*Algorithme 12 : Fonction **estVide** pour un réservoir - version 1.0*

© ENI Editions - All rigths reserved

```
Algorithme estVide
    # r est-il vide ?
Entrée
    r : RÉSERVOIR
Résultat : BOOLÉEN
précondition
    0 ≤ r.contenance ≤ r.capacité
réalisation
    Résultat <- (r.contenance = 0)
postcondition
    Résultat  = (r.contenance = 0)
    0 ≤ r.contenance ≤ r.capacité
fin estVide
```

(`r.contenance = 0`) est un prédicat : il prend la valeur VRAI ou FAUX selon que le réservoir a une contenance nulle ou non. Cette valeur sert à initialiser la variable **Résultat** calculée par la fonction. Ainsi, ce résultat est VRAI si et seulement si la contenance actuelle du réservoir est nulle et alors le réservoir est vide. Lorsque la contenance n'est pas nulle, **Résultat** prend la valeur FAUX, ce qui indique que le réservoir n'est pas vide.

La fonction **estPlein** est définie de la même façon.

*Algorithme 13 : Fonction **estPlein** pour un réservoir - version 1.0*

```
Algorithme estPlein
    # r est-il plein ?
Entrée
    r : RÉSERVOIR
Résultat : BOOLÉEN
précondition
    0 ≤ r.contenance ≤ r.capacité
réalisation
    Résultat <- (r.contenance = r.capacité)
postcondition
    Résultat = (r.contenance = r.capacité)
    0 ≤ r.contenance ≤ r.capacité
fin estPlein
```

Le réservoir est plein si et seulement si sa contenance est égale à sa capacité. C'est ce qu'exprime l'affectation de la variable **Résultat** qui prend pour valeur celle du prédicat à droite du symbole d'affectation.

Remarquez aussi dans ces deux fonctions la première clause de la postcondition. Cette clause prend la valeur VRAI ou FAUX selon que **Résultat** a la valeur de l'expression booléenne à droite du signe d'égalité. Lorsque **Résultat** a la même valeur que l'expression, c'est que l'algorithme rend une valeur correcte. Sinon, c'est que l'algorithme ne fait pas un calcul correct et il convient alors de le corriger. Encore une fois, la signature de la fonction, complétée de la précondition et de la postcondition suffit à la définir.

Les deux fonctions **estVide** et **estPlein** sont deux prédicats qui admettent la même précondition. On peut remarquer que cette précondition caractérise un réservoir en général et non pas seulement les données en entrée ou le résultat de ces prédicats. En particulier, la condition qu'elle exprime est encore vraie lorsque l'algorithme se termine et retourne son résultat. C'est ce qui est dit dans la postcondition où on retrouve la clause `0 ≤ r.contenance ≤ r.capacité`. En effet, pour tout réservoir on a :

`0 ≤ contenance ≤ capacité`

et cette condition doit rester toujours vraie, dans tous les états du système logiciel. C'est ce que l'on appelle un *invariant* de type. Cet invariant caractérise un réservoir, quel qu'il soit et à tout instant de son cycle de vie, pour tout état du système. Pour marquer qu'il s'agit là d'une propriété permanente, qui ne dépend pas de l'état d'un réservoir particulier, mais qui caractérise le type **RÉSERVOIR**, on la précise dans la définition du type, comme ceci :

```
type
    RÉSERVOIR
structure
    capacité   : RÉEL
    contenance : RÉEL
invariant
    0 ≤ contenance ≤ capacité
fin RÉSERVOIR
```


© ENI Editions - All rigths reserved

Cette condition doit toujours être vérifiée, dans la précondition, pendant l'exécution de l'algorithme et dans la postcondition. Il est alors inutile de la préciser dans la précondition et la postcondition. Il est implicite que l'invariant étant une caractéristique intrinsèque des données du type défini, il reste toujours vrai et doit être vérifié dans tous les états du système. On peut reprendre la définition du prédicat **estVide** comme dans l'algorithme suivant :

*Algorithme 14 : Fonction **estVide** pour un réservoir - version 2.0*

```
Algorithme estVide
    # r est-il vide ?
Entrée
    r : RÉSERVOIR
Résultat : BOOLÉEN
précondition
    aucune
réalisation
    Résultat <- (r.contenance = 0)
postcondition
    Résultat  = (r.contenance = 0)
fin estVide
```

Bien entendu, la précondition et la postcondition complètes sont celles de la version 1.0 étudiée plus haut.

Les commandes **ajouter** ou **enlever** ajoute ou enlève une quantité donnée de liquide au contenu actuel du réservoir pour augmenter ou diminuer respectivement sa contenance. On peut remarquer que **ajouter** consiste à *augmenter positivement* la contenance alors que **enlever** consiste à l'*augmenter négativement*. On a donc :

ajouter(r,quantité:**RÉEL**)::=**augmenter**(r, quantité) avec quantité ≥ 0
enlever(r,quantité:**RÉEL**)::=**augmenter**(r, -quantité) avec quantité ≥ 0

Cette remarque permet de définir la procédure ajouter :

Algorithme 15 : Procédure ajouter à un réservoir

```
Algorithme ajouter
    # Ajouter quantité à la contenance du réservoir r.
Entrée
    r        : RÉSERVOIR
    quantité : RÉEL
précondition
    0 ≤ quantité ≤ r.capacité - r.contenance
réalisation
    augmenter(r, quantité)
postcondition
    r.contenance = ancien(r.contenance) + quantité
fin ajouter
```

Le sous-algorithme **augmenter** est une procédure qui factorise les instructions communes aux algorithmes **ajouter** et **enlever**. Il est défini ci-dessous.

Algorithme 16 : Augmenter la contenance du réservoir

```
Algorithme augmenter
    # Augmenter la contenance du réservoir r de quantité.
Entrée
    r        : RÉSERVOIR
    quantité : RÉEL
réalisation
    r.contenance <- r.contenance + quantité
postcondition
    r.contenance = ancien(r.contenance) + quantité
fin augmenter
```

Il n'y a pas de précondition à cette procédure, si ce n'est la clause d'invariant, bien entendu. En particulier, `quantité`

© ENI Editions - All rigths reserved

n'est plus assujettie à être positive. En effet, enlever une quantité positive de liquide, c'est augmenter la contenance de l'opposé de cette quantité. La procédure **enlever** est donc définie par :

Algorithme 17 : Procédure enlever à un réservoir

```
Algorithme enlever
    # Enlever quantité à la contenance du réservoir r.
Entrée
    r        : RÉSERVOIR
    quantité : RÉEL
précondition
    0 ≤ quantité ≤ r.contenance
réalisation
    augmenter(r, -quantité)
postcondition
    r.contenance = ancien(r.contenance) - quantité
fin enlever
```

Comme cela a déjà été dit plusieurs fois, la responsabilité d'une commande est de modifier l'état du système. La procédure **ajouter**, par exemple, modifie la contenance du réservoir. Il faut alors s'assurer que le nouvel état obtenu reste cohérent avec la définition d'un réservoir. Ici, il convient de s'assurer que l'invariant de type est respecté. Voyons cela.

Initialement on a :

`(0 ≤ quantité ≤ r.capacité - r.contenance)` et `(0 ≤ r.contenance ≤ r.capacité)`

C'est la précondition, dont on exige qu'elle soit vérifiée avant l'exécution de l'algorithme. Cette précondition est un prédicat sur l'état du réservoir caractérisé par les valeurs des variables en entrée. Lorsque la réalisation de l'algorithme est faite, on a :

`(r.contenance = ancien(r.contenance) + quantité)`

et

`(0 ≤ quantité ≤ r.capacité - ancien(r.contenance))`

En ajoutant `ancien(r.contenance)` aux deux membres de la deuxième inégalité, on obtient :

```
ancien(r.contenance) + quantité ≤  ancien(r.contenance) + r.capacité -
                                   ancien(r.contenance) + quantité
```

D'où, finalement :

```
ancien(r.contenance) + quantité = r.contenance ≤ r.capacité
```

Ce qui termine la preuve que l'algorithme maintient l'invariant.

La commande **enlever** réalise l'opération duale de **ajouter** et on démontre de la même façon qu'elle maintient l'invariant.

*Exercice 3 : **enlever** maintient l'invariant de type*

*Démontrer que la procédure **enlever** maintient l'invariant du type **RÉSERVOIR**.*

Les opérations **remplir** et **vider** utilisent les algorithmes précédents. Voyons cela.

Vider un réservoir consiste à **enlever** sa contenance. D'où la définition de l'algorithme de cette procédure :

Algorithme 18 : vider un réservoir

```
Algorithme vider
    # Vider le réservoir r
Entrée
    r : RÉSERVOIR
précondition
    aucune
réalisation
    enlever(r, r.contenance)
```


© ENI Editions - All rigths reserved

```
postcondition
    estVide(r)
fin vider
```

La réalisation utilise l'algorithme **enlever** défini précédemment. Il convient de remarquer surtout que la postcondition utilise le prédicat **estVide**. Plus généralement, la précondition ou la postcondition d'un algorithme peut utiliser toute instruction qui ne modifie pas l'état du système logiciel. Par conséquent, les paramètres sont utilisables, comme cela a déjà été vu, mais aussi toute fonction définie qui ne fait appel qu'aux ressources locales au module (à quelques réserves près qui seront étudiées plus tard) dont on veut exprimer les spécifications.

Remplir un réservoir, c'est lui ajouter une quantité égale à la différence entre sa capacité et sa contenance. Le réservoir est alors plein. Il n'est donc pas difficile d'écrire cet algorithme.

*Exercice 4 : Définition de la procédure **remplir***

*Écrire la définition complète de l'algorithme **remplir**.*

Pour être complet, il reste à démontrer que ces algorithmes sont corrects. Cette correction suppose essentiellement le maintien de l'invariant de type, mais elle n'est pas plus difficile à établir que celle de **ajouter**.

Exercice 5 : Correction des algorithmes précédents

*Démontrer que les algorithmes **remplir** et **vider** maintiennent l'invariant du type **RÉSERVOIR**.*

Le diagramme d'états d'un réservoir n'est pas difficile à obtenir. Il est fait de trois états fondamentaux : VIDE, PLEIN et un état dans lequel le réservoir n'est ni vide ni plein.

Exercice 6 : Diagramme d'états d'un réservoir

Construire le diagramme d'états d'un réservoir.

Le dernier exercice résolu de cette section est d'abord déroutant. Il semble difficile, mais il l'est moins qu'il n'y paraît. Il n'est pas entièrement résolu, mais ce chapitre a déjà abordé quasiment toutes les connaissances nécessaires pour en donner une solution complète.

Exercice résolu 13 : Le jour du premier Mai

Écrire un algorithme calculant, pour une année dont on donne le millésime, le jour de la semaine où tombe le premier Mai.

Ainsi, par exemple, le 1er Mai 2005 tombait un dimanche. Le 1er Mai 1983 était aussi un dimanche. Le 1er Mai 2001 était un mardi.

Solution partielle

La solution présentée est partielle. Elle sera complétée dans les chapitres suivants.

Les données du problème sont constituées, d'une part, d'un millésime et, d'autre part, d'une date invariable : le 1er Mai. Le millésime peut, en toute généralité, être un entier relatif quelconque, puisqu'il est possible de faire référence à une année avant JC. Cependant, cela n'a pas beaucoup de sens, ne serait-ce que parce que la date du 1er Mai célèbre un événement récent, du dix-neuvième siècle. Il faut donc préciser davantage le problème. Ce sera fait plus bas.

Le résultat du calcul demandé est le jour de la semaine où tombe le 1er Mai d'une année donnée. Ce résultat a sept valeurs possibles : lundi, mardi, ..., dimanche. L'énoncé donne que le 1er Mai 1983 était un dimanche. Comme un an a 365 jours, mais 52 semaines qui font 52 x 7 = 364 jours, chaque année qui passe introduit un décalage de 1 jour supplémentaire par rapport à l'année précédente. Ainsi, en notant **successeur**(j) le jour de la semaine qui suit un jour j quelconque, on a :

1er Mai 1983 : dimanche ;

1er Mai 1984 : **successeur**(dimanche) = lundi

En vérifiant sur un vieux calendrier, on constate que le 1er Mai 1984 était un mardi et non pas un lundi. C'est que 1984 est une année bissextile, qui introduit un décalage supplémentaire de 1 jour pour le 29 février. On a donc :

1er Mai 1983 : dimanche ;

1er Mai 1984 : **successeur**(**successeur**(dimanche)) = mardi

Nous disposons donc d'un moyen d'atteindre le résultat :

1. on se donne une année de référence dont on connaît le jour du 1er Mai ;

2. pour une année donnée, on calcule le nombre de jours de décalage par rapport au jour du 1er Mai de l'année de référence.

© ENI Editions - All rigths reserved

Nous choisirons une année reculée, qui constituera la borne inférieure du domaine des millésimes et permettra donc d'exprimer la précondition sur cette donnée.

Pour une année reculée dans le temps, on peut avoir un décalage de plusieurs jours. Il faut donc pouvoir cumuler les décalages. Pour simplifier, notons **s**(j) le successeur d'un jour j quelconque de la semaine et $\mathbf{s}^k$(j) l'application de l'opération successeur à k reprises. Ainsi, pour l'exemple qui précède, on obtient :

1er Mai 1983 : dimanche ;

1er Mai 1984 : $\mathbf{s}^2$(dimanche) = mardi

Dans la suite, $\mathbf{s}^k$(j) sera aussi noté **s**(j,k), comme dans **s**(dimanche, 2) pour $\mathbf{s}^2$(dimanche).

- Posons **travail**(millésime) le jour du 1er Mai de l'année millésime. Ainsi, **travail**(1983) = dimanche. À l'année millésime + 1 on a :

```
travail(millésime+1) = s(travail(millésime), 1) si millésime+1 n'est
pas bissextile ;
travail(millésime+1) = s(travail(millésime), 2) si millésime+1 est
bissextile ;
```

Pour un décalage d'un nombre d'années égal à `an`, on aura :

```
travail(millésime + an) = s(travail(millésime), an+bis)
```

formule dans laquelle `bis` désigne le nombre d'années bissextiles pendant `an` années.

- Comme il y a une année bissextile tous les quatre ans, on devrait trouver :

```
bis = quotient(an, 4)
```

Cependant, en reprenant l'exemple de 1983, on obtient pour 1984 :

```
an  = 1
bis = quotient(1, 4) = 0
```

Or, 1984 étant bissextile, on devrait obtenir `bis=1`. Ainsi, `bis = quotient(an, 4)` uniquement si un décalage de 1, 2 ou 3 jours ne donne pas d'année bissextile, donc si les années millésime+1, millésime+2 et millésime+3 ne sont pas bissextiles. Comme il y a une année bissextile tous les quatre ans « à quelque chose près », il suffit de choisir une année de référence bissextile.

La répartition des années bissextiles présente des irrégularités. Les années qui marquent une limite de siècle, comme 2100 par exemple, ne sont pas bissextiles, mais 2000 l'était. 2000 n'introduisait pas d'irrégularité. En fait, les années qui marquent la limite d'un siècle ne sont pas bissextiles, sauf celles dont les deux premiers chiffres forment un nombre multiple de 4. Autrement dit un millésime multiple de 400 est une année bissextile. Ainsi, 1900 et 2100 ne sont pas bissextiles car 19 et 21 ne sont pas des multiples de 4, ou encore 1900 et 2100 ne sont pas des multiples de 400, alors que 2000 est bissextile puisque 2000 est un multiple de 400 ou encore 20 est un multiple de 4.

On peut pourtant choisir 1900 comme année de référence puisque l'année bissextile suivante sera 1904 sans irrégularité dans la succession d'une année bissextile tous les 4 ans jusqu'en 2100. Convenons donc, pour simplifier :

- 1900 sera l'année de référence ;

- 2100 sera la limite supérieure des millésimes acceptables.

Il devient alors possible de donner une spécification partielle de l'algorithme :

```
Entrée
    millésime : ENTIER
    Résultat  : ???
précondition
    1900 ≤ millésime < 2100
postcondition
    Résultat = le jour de la semaine où tombe le 1er Mai
    de l'année millésime
```

Il nous faut connaître le jour du 1er Mai 1900 : c'était un mardi. D'où une première version de l'algorithme demandé, qui

© ENI Editions - All rigths reserved

utilise un nouveau type de données défini par :

```
type JOUR
    (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche)
fin JOUR
```

Algorithme 19 : Jour du 1er Mai - version 1.0

```
Algorithme travail
Entrée
    millésime : ENTIER
Résultat : JOUR
précondition
    1900 ≤ millésime < 2100
variable
    an  : ENTIER # décalage depuis 1900.
    bis : ENTIER # nombre d'années bissextiles depuis 1900.
réalisation
    an <- millésime - 1900
    bis <- quotient(an, 4)
    Résultat <- s(mardi, an+bis)
postcondition
    Résultat = le jour de la semaine où tombe le 1er Mai
    de l'année millésime
fin travail
```

Essayons sur un exemple avec millésime = 1983 :

```
an  = 1983 - 1900 = 83
bis = quotient(an, 4) = quotient(83, 4) = 20
Résultat = s(mardi, 83+20) = s(mardi, 103)
```

Il reste à définir l'opérateur **successeur** ou **s**. L'algorithme suivant spécifie cet opérateur. Sa définition complète et sa réalisation seront étudiées au chapitre qui traite de l'itération.

*Algorithme 20 : Spécification de l'opérateur **successeur** - version 1.0*

```
Algorithme successeur, s
    # Le « décalage »-ème successeur du jour j.
Entrée
    décalage : ENTIER
précondition
    décalage ≥ 0
postcondition
    Résultat = le jour de la semaine égal à 'j' décalé
    d'un nombre de jours égal à 'décalage'
fin successeur
```

Remarquez que cette spécification définit aussi **s** comme alias pour le nom **successeur** de la procédure. Ce n'est qu'une facilité pour réduire la longueur du nom en définissant un substitut plus court.

Pour obtenir une solution complète avec ce que nous savons, on peut reconsidérer la représentation des données. Ainsi, par exemple, au lieu d'utiliser un type **JOUR** pour les jours de la semaine, il est possible de définir un codage des jours en nombres entiers, comme ceci :

```
(lundi=1, mardi=2, mercredi=3, jeudi=4, vendredi=5, samedi=6, dimanche=7)
```

Comme le successeur de dimanche est lundi, on doit modifier notre arithmétique de sorte que **s**(dimanche, 1) = lundi, ou, puisque nous utilisons des nombres entiers, dimanche + 1 = lundi. Comme lundi=1 par convention, ceci nous conduit à poser dimanche=0 :

```
(lundi=1, mardi=2, mercredi=3, jeudi=4, vendredi=5, samedi=6, dimanche=0)
```

Notre arithmétique doit donc donner samedi + 1 = dimanche. Or samedi = 6 et dimanche = 0. L'arithmétique utilisée doit donc donner 6 + 1 = 0, c'est-à-dire le reste de la division de 6 + 1 par 7. Ainsi, en désignant par jour un jour quelconque

© ENI Editions - All rigths reserved

de la semaine, on aura :

```
successeur(jour, 1) = reste(jour + 1, 7)
```

Pour 1983, le jour du 1er Mai s'est décalé de plusieurs semaines depuis le 1er Mai 1900. Le nombre de semaines est le quotient entier de la division de `décalage` par 7. Le reste de cette division donne le jour de la semaine correspondant. Pour 1983, le décalage total est 2+103=105 puisque mardi=2. On obtient **reste**(105, 7) = 0. Par conséquent, le jour cherché est le jour de code 0, soit un dimanche. L'exercice suivant propose d'utiliser ce codage pour reprendre la solution du problème. On obtient ainsi la solution présentée par la référence [ARS80].

Exercice 7 : Jour du 1er Mai - version 2.0

La fonction successeur n'est pas nécessaire pour résoudre le problème.

Refaire la version 1.0 de cet algorithme en représentant les jours de la semaine par leur code numérique.

La section suivante propose quelques exercices pour mettre en pratique ce qui a été vu dans les sections précédentes.

© ENI Editions - All rigths reserved

Exercices

Les solutions des exercices proposés ne font intervenir aucune notion nouvelle. Seules les quelques notions étudiées dans ce chapitre sont nécessaires pour donner une solution complète à ces exercices, même si, parfois, il existe une solution meilleure que celle actuellement accessible. Chaque exercice doit être entièrement spécifié, la réalisation de l'algorithme proposé doit être complète et, s'il y a lieu, démontrée. Certains énoncés sont volontairement imprécis et donc permettent, parfois, différentes interprétations. Les algorithmes, pour les résoudre, doivent être parfaitement définis et ne laisser subsister aucune ambiguïté. Il faudra donc faire des choix qui sont laissés à la discrétion du lecteur, mais qui doivent être clairement énoncés, si ce n'est justifiés.

Exercice 8 : Taux, TVA et placements

1. Écrire un algorithme qui calcule le prix toutes taxes comprises (TTC) pour un prix hors taxe et un taux de TVA donné.

2. Écrire un algorithme qui calcule le montant des intérêts rapportés par un capital placé à un taux donné pendant une durée donnée, exprimée en mois.

Exercice 9 : Moyenne arithmétique pondérée

1. Écrire un algorithme qui calcule la moyenne arithmétique de trois nombres donnés.

2. Même question pour une moyenne pondérée quand on donne les nombres et les coefficients de pondération.

Exercice 10 : Aire du triangle

1. Écrire un algorithme qui calcule l'aire d'un triangle dont on donne la mesure d'un côté et celle de la hauteur relative à ce côté.

2. Cet algorithme est-il utilisable pour un triangle rectangle dont on donne les mesures des deux côtés perpendiculaires ?

Exercice 11 : Salaire et heures supplémentaires

Le calcul d'un bulletin de salaire prend en compte le salaire brut associé aux heures « normales » dues par le salarié et les heures « supplémentaires » travaillées dans le mois. Les heures supplémentaires sont rémunérées selon les règles de gestion suivantes :

- *taux horaire majoré de 125 % pour les heures de la 36ème à la 43ème ;*
- *taux horaire majoré de 150 % pour les heures à partir de la 44ème.*

La majoration intervient sur le taux horaire normal, calculé à partir du salaire mensuel brut pour une année de 52 semaines réparties sur 12 mois, sur la base de 35 heures travaillées hebdomadaires.

Écrire l'algorithme qui calcule le montant des heures supplémentaires à rémunérer, à partir du salaire mensuel brut et du nombre d'heures supplémentaires.

On pourra supposer que le calcul est toujours utilisé pour un nombre d'heures supérieur à 8. Le problème général suppose l'étude préalable du chapitre suivant qui traite de l'alternative.

Exercice 12 : Compte de dépôt

On considère les comptes de dépôts hébergés par une banque pour ses clients. Un retrait n'est autorisé que si le solde du compte reste non négatif.

*1. Définir le type de données **COMPTE**.*

2. Définir les opérations applicables.

Dans certaines circonstances et pour certains clients, la banque autorise un découvert limité et temporaire.

3. Refaire les définitions précédentes pour permettre ces découverts.

© ENI Editions - All rigths reserved

Notes bibliographiques

La définition d'un algorithme est difficile quand on veut rester à un niveau élémentaire. Une excellente référence, ancienne mais dont la lecture reste un vrai bonheur est [ARS80]. Les exercices de la section Exercices résolus, excepté celui sur les réservoirs, sont inspirés de ce petit livre. La spécification d'un algorithme, telle qu'elle est présentée ici, vient de la programmation par contrat inventée par Bertrand MEYER. Les références sur ce sujet sont nombreuses. Le livre [MEY00] de cet auteur traite de la conception et de la programmation objet et fournit une bibliographie complète sur la programmation par contrat. Les recettes de cuisine du début du chapitre sont extraites de [MAR74].

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a abordé l'écriture d'algorithmes. Un algorithme est la description d'un procédé de calcul composé d'une succession d'opérations. Une fonction calcule un résultat à partir de données qu'elle ne modifie pas. Une procédure modifie un état en calculant les nouvelles valeurs des données.

L'élaboration d'un algorithme se décompose en deux phases qui ne doivent pas être confondues. La spécification précise ce que fait l'algorithme sans dire comment il le fait. La spécification d'un algorithme fait partie de sa documentation. Elle est composée de la signature du module logiciel, de la précondition et de la postcondition. La définition de l'algorithme exprime par des instructions comment il réalise son calcul.

La précondition regroupe les conditions que doivent satisfaire les données en entrée pour que l'algorithme calcule ses résultats. Elles doivent être simultanément VRAIEs pour que l'algorithme effectue un calcul « juste » pour produire les résultats attendus. C'est de la responsabilité du client logiciel des services de l'algorithme d'assurer la réalisation de ces conditions. La postcondition exprime les garanties qu'apporte l'algorithme sur le travail qu'il réalise et les résultats qu'il produit, dès lors que la précondition est satisfaite.

La syntaxe pour écrire un algorithme n'est pas contrainte. Il n'existe pas de « langage algorithmique » et l'auteur d'un algorithme est entièrement libre d'utiliser la « grammaire » qui lui convient. Les seules exigences du domaine sont une lisibilité immédiate des instructions, une expression sans ambiguïté des opérations utilisées et une forme d'expression qui permette de se convaincre que l'algorithme fait bien ce qu'il annonce et seulement cela. En particulier, il n'est pas nécessaire d'imiter la grammaire d'un langage de programmation des ordinateurs même si, *in fine*, l'algorithmique ne s'est développée que pour et par le traitement automatique de l'information que permettent les calculateurs programmables.

© ENI Editions - All rigths reserved

Bibliographie

[ARS77] Jacques ARSAC : *La construction de programmes structurés* ; DUNOD Informatique - Phase formation, PARIS, 1977.

[ARS80] Jacques ARSAC : *Premières leçons de programmation* ; CEDRIC/NATHAN, PARIS, 1980.

[CB84] G. CLAVEL, J. BIONDI : *Introduction à la programmation - Tome 2 : Structures de données* ; MASSON, 1984.

[MAR74] Tante Marie : *La véritable cuisine de famille* ; Éditions A. TARIDE, 1974.

[MEY00] Bertrand MEYER : *Conception et programmation orientées objet* ; EYROLLES, PARIS, 2000.

© ENI Editions - All rigths reserved

Introduction

Ce chapitre présente l'une des constructions fondamentales de l'algorithmique : « la structure de choix » ou encore « l'alternative ». Comme son nom l'indique, elle permet de procéder au choix d'un traitement, parmi plusieurs possibles, selon des conditions établies en fonction du contexte et des données à traiter.

La deuxième section présente l'alternative, à l'aide d'un exemple trivial décliné en plusieurs versions. Le but est d'exposer l'utilisation de la nouvelle construction en précisant le vocabulaire. La troisième section présente des exercices résolus et la dernière des exercices d'application.

© ENI Editions - All rigths reserved

Définition de l'alternative

Cette section ne propose pas de définition formelle de l'alternative. Elle ne fait que la présenter, à l'aide d'un exemple dont les différentes versions permettent de préciser quelques points de méthode.

Exemple

On considère deux données a et b d'un type **T** quelconque, mais **COMPARABLES**. Autrement dit, les données de type **T** sont mutuellement comparables. Il existe donc une relation d'ordre total, notée ≤, sur les données de type **T**. Cela signifie que, étant donnés deux éléments a et b de type **T**, il est toujours possible de les comparer, par exemple pour désigner celui qui est inférieur ou égal à l'autre. a et b peuvent être des nombres, comme des entiers par exemple, mais pas seulement. Ils peuvent aussi être des caractères ou des chaînes de caractères, comme des phrases du langage courant par exemple, puisque l'ordre alphabétique, entre autres, permet de les comparer. Pour indiquer que le type **T** est quelconque, mais que les données de type **T** sont toutes comparables mutuellement, on notera, par convention :

```
a, b : T -> COMPARABLE
```

On indique ainsi que le type **T***dérive* du type **COMPARABLE**, ou que toute donnée de type **T** est aussi de type **COMPARABLE**. Une autre façon de noter la même idée est de préciser que a et b appartiennent au type (**T**, ≤) :

```
a, b : (T, ≤)
```

On veut écrire un algorithme qui classe a et b en ordre croissant.

Il s'agit d'écrire la définition d'une suite d'instructions qui laissent a et b tels que a ≤ b. On peut donc déjà donner la spécification de l'algorithme avec ce que nous avons appris au chapitre Programmes directs. L'algorithme ci-dessous donne cette spécification.

Algorithme 1 : Spécification du classement de deux données comparables

```
Algorithme classer
    # Classe a et b en ordre croissant.
Entrée
    a, b : T -> COMPARABLE
précondition
    aucune
postcondition
    a ≤ b
fin classer
```

Dans la réalisation de cet algorithme, seule l'opération de classement des deux nombres apporte quelque chose de nouveau. Détaillons-la :

```
# Classement de deux données comparables en ordre croissant.
si
    a > b
alors
    # a > b : replacer a et b en ordre.
    échanger(a, b)
    # a ≤ b : a et b sont en ordre.
sinon
    rien
fin si
```

Cette forme est simple : si la première donnée, a, est strictement supérieure à la seconde, b, elles sont replacées en ordre en échangeant leur valeur. C'est ce que fait l'instruction qui commence à **alors** et se termine avec le mot **sinon**. Dans le cas contraire, les données sont déjà en ordre et il n'y a rien à faire. C'est ce qui est indiqué par l'instruction qui commence à **sinon** et qui se termine par l'expression **fin si**.

Lorsque la condition booléenne exprimée par le bloc qui commence au mot **si** prend la valeur VRAI, ce sont les instructions du bloc qui commencent au mot **alors** et qui se terminent avant **sinon** qui sont exécutées. Lorsque cette condition prend la valeur FAUX, ce sont les instructions du bloc qui commencent au mot **sinon** et qui se terminent avec l'expression **fin si** qui sont exécutées. Dans tous les cas, seules les instructions de l'un des deux blocs sont exécutées : il s'agit bien d'une *alternative*.

Ce qui est nouveau ici, c'est la construction :

© ENI Editions - All rigths reserved

```
si
    <expression booléenne>
alors
    <traitements 1>
sinon
    <traitements 2>
fin si
```

<expression booléenne> est une expression prenant ses valeurs dans l'ensemble {VRAI ; FAUX}. Lorsque <expression booléenne> prend la valeur VRAI, seules les instructions de l'alternant décrit entre **alors** et **sinon** sont exécutées. Ici, ce sont les instructions qui constituent le bloc <traitements 1>. Lorsque <expression booléenne> prend la valeur FAUX, ce sont les instructions décrites entre **sinon** et **fin si**, c'est-à-dire les instructions du bloc <traitements 2>, qui sont exécutées. Dans tous les cas, les traitements reprennent ensuite en séquence aux instructions qui suivent **fin si**.

La première version complète de cet algorithme devient :

Algorithme 2 : Classer deux données comparables en ordre croissant - version 1.0

```
Algorithme classer
    # Classe a et b en ordre croissant.
Entrée
    a, b : T → COMPARABLE
précondition
    aucune
réalisation
    si
        a > b
    alors
        # a > b : replacer a et b en ordre.
        échanger(a, b)
        # a ≤ b : a et b sont en ordre.
    sinon
        rien
    fin si
postcondition
    a ≤ b
fin classer
```

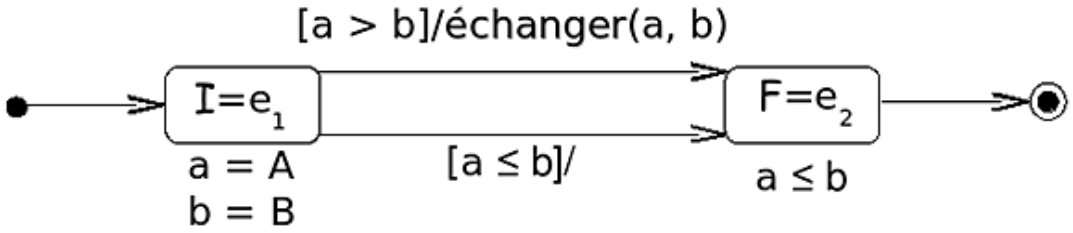
Lorsqu'un alternant ne contient aucune instruction, comme le bloc **sinon** ci-dessus, il n'est pas nécessaire de le détailler. Ainsi, on écrira plutôt l'algorithme suivant :

Algorithme 3 : Classer deux données comparables en ordre croissant - version 2.0

```
Algorithme classer
    # Classe a et b en ordre croissant.
Entrée
    a, b : T -> COMPARABLE
précondition
    aucune
réalisation
    si
        a > b
    alors
        # a > b : replacer a et b en ordre.
        échanger(a, b)
        # a ≤ b : a et b sont en ordre.
    fin si
postcondition
    a ≤ b
fin classer
```

Cet algorithme réalise le tri de deux données comparables en ordre croissant. Il est facile de s'en convaincre en suivant l'évolution des états successifs du système constitué des valeurs des variables a et b. Ces états sont décrits, dans le texte de l'algorithme, par les commentaires qui accompagnent les instructions qui les modifient. Il est aussi possible de décrire la succession des états par un diagramme. C'est fait sur la figure suivante :

© ENI Editions - All rigths reserved



Les transitions entre états sont représentées par des flèches. Chacune porte une étiquette de la forme `événement [garde]/action`. Lorsque l'`événement` se produit, l'expression booléenne `garde` est évaluée. Si le résultat de cette évaluation est VRAI, l'`action` est exécutée et elle provoque la transition entre les deux états concernés. Sur le diagramme de la figure ci-dessus, l'`événement` associé à chacune des deux transitions est vide et, par conséquent, c'est l'entrée dans l'état $I=e_1$ qui provoque immédiatement l'évaluation de la `garde` sur la première transition. Pour la deuxième transition, seule la `garde` est définie. Dans ce cas, aucune action particulière n'est entreprise lors de la transition lorsque la `garde` est évaluée à VRAI.

Exemple

Soit n un nombre d'un ensemble quelconque. Quel est le signe de ce nombre ?

On peut spécifier simplement ce problème. C'est fait par l'algorithme suivant :

Algorithme 4 : Spécification du signe d'un nombre - version 1.0

```
Algorithme signe1
    # Calcule le signe de n.
Entrée
    n : RÉEL
Résultat : ENTIER
précondition
    aucune
postcondition
    n < 0 => Résultat = -1
    n = 0 => Résultat =  0
    n > 0 => Résultat = +1
fin signe1
```

Ainsi, cette fonction rend -1 si le paramètre n est strictement négatif, 0 s'il est nul ou +1 s'il est strictement positif. Ces valeurs sont choisies par convention. On peut en choisir d'autres et ce serait légitime. -1, 0 et +1 sont les valeurs habituellement utilisées dans ce cas.

Remarquez aussi la postcondition. Elle est faite de trois clauses dont la conjonction doit prendre la valeur VRAI lorsque l'algorithme est correct. Autrement dit, la postcondition est l'expression booléenne constituée des trois clauses connectées par l'opérateur **et** logique qui reste, ici, implicite :

```
    n < 0 => Résultat = -1
  et
    n = 0 => Résultat =  0
  et
    n > 0 => Résultat = +1
```

L'expression de la postcondition dans l'algorithme précédent est déroutante au premier abord et il n'est pas immédiatement clair qu'une conjonction des trois clauses exprime ce que fait cet algorithme. La justification fait appel aux Mathématiques et à l'algèbre de Boole. Les lecteurs qui ne seraient pas intéressés par la justification de cette postcondition peuvent passer directement à la suite de la description de l'algorithme.

La première clause est une implication : n < 0 => **Résultat** = -1. L'implication (a => b) est définie par (**non** a **ou** b). La clause (n < 0 => **Résultat** = -1) est donc équivalente à (n ≥ 0 **ou Résultat** = -1). Par conséquent, cette première clause rend toujours le résultat VRAI, sauf lorsque (n < 0 **et Résultat** ≠ -1). Le même raisonnement montre que la seconde clause de la postcondition rend toujours VRAI, sauf lorsque (n = 0 **et Résultat** ≠ 0). Enfin, la troisième clause rend toujours VRAI, sauf dans le cas (n > 0 **et Résultat** ≠ +1). Ainsi, pour les 12 combinaisons possibles de valeurs sur le signe de n et de **Résultat**, seule la disjonction des trois clauses exhibées ici rend un résultat FAUX. Autrement dit, la postcondition prend la valeur FAUX lorsque :

(n < 0 et **Résultat** ≠ -1) **ou** (n = 0 **et Résultat** ≠ 0) **ou** (n > 0 **et Résultat** ≠ +1)

© ENI Editions - All rigths reserved

et elle prend la valeur VRAI dans tous les autres cas :

(n < 0 => **Résultat** = -1) **et** (n = 0 => **Résultat** = 0) **et** (n > 0 => **Résultat** = +1)

Revenons à l'algorithme. Le résultat sera obtenu en comparant `n` à 0. L'algorithme consiste donc à comparer deux nombres, ici `n` et 0. On obtient un algorithme plus général en comparant deux données comparables quelconques `n` et `m`.

Algorithme 5 : Spécification de la comparaison de deux données comparables - version 1.0

```
Algorithme comparer
    # Comparer n à m.
Entrée
    n, m : T -> COMPARABLE
Résultat : ENTIER
précondition
    aucune
postcondition
    n < m => Résultat = -1
    n = m => Résultat =  0
    n > m => Résultat = +1
fin comparer
```

Obtenir le signe d'un nombre selon **signe1** consistera à utiliser **comparer** avec m = 0. Il est donc possible d'écrire complètement la fonction **signe** :

Algorithme 6 : Signe d'un nombre - version 2.0

```
Algorithme signe
    # Calcule le signe de n.
Entrée
    n : RÉEL
Résultat : ENTIER
précondition
    aucune
réalisation
    Résultat <- comparer(n, 0)
postcondition
    n < 0 => Résultat = -1
    n = 0 => Résultat =  0
    n > 0 => Résultat = +1
fin signe
```

Cette version du signe d'un nombre utilise l'algorithme **comparer** et il reste à l'écrire. Une première version est celle de l'algorithme suivant :

Algorithme 7 : Réalisation de la comparaison de deux données comparables - version 1.0

```
réalisation
    si
        n < m
    alors
        # n < m : rendre -1.
        Résultat <- -1
    sinon
        # n ≥ m : discriminer les deux cas possibles.
        si
            n = m
        alors
            # Égalité : rendre 0
            Résultat <- 0
        sinon
            # n > m :  rendre +1
            Résultat <- +1
        fin si
```


© ENI Editions - All rigths reserved

```
    fin si
```

Cette réalisation est facile à comprendre. Quand `n < m`, le résultat est -1. Sinon, c'est que `n ≥ m`. Il faut alors distinguer le cas `n = m` du cas `n > m`. C'est l'objet de la nouvelle alternative imbriquée dans l'alternant **sinon** de la première. Il est aussi possible d'écrire cet algorithme comme ceci :

Algorithme 8 : Réalisation de la comparaison de deux données comparables - version 2.0

```
si n < m alors Résultat <- -1 fin si
si n = m alors Résultat <-  0 fin si
si n > m alors Résultat <- +1 fin si
```

Dans cette version, les trois tests sont réalisés. Même quand l'algorithme a reconnu que n < m par exemple et que **Résultat** a pris la valeur -1, il teste encore l'égalité n = m et échoue, puis il teste n > m et échoue. Ainsi, cette solution, certainement correcte, fait le travail de détermination du signe d'une façon inefficace. *Quelle solution de la réalisation convient-il de retenir ?* Celle de la version 1.0 fait moins de comparaisons mais, toute proportion gardée, est moins lisible que la version 2.0. Celle-là au contraire est facile à lire et à comprendre d'emblée, mais elle fait plus de comparaisons et les deux tiers d'entre elles sont inutiles. Quelle que soit le solution retenue, elle est spécifiée de la même façon par l'algorithme 4.

La réalisation de l'algorithme 7 peut encore être écrite d'une façon un peu différente dans les détails afin d'améliorer sa lisibilité. L'algorithme ci-dessous reprend la solution complète du problème.

Algorithme 9 : Comparaison de deux données comparables - version 3.0

```
Algorithme comparer
    # Comparer n à m.
Entrée
    n, m : T -> COMPARABLE
Résultat : ENTIER
précondition
    aucune
réalisation
    si n < m alors
        # n < m : rendre -1.
        Résultat <- -1
    sinon si n = m alors
        # Égalité : rendre 0
        Résultat <- 0
    sinon
        # n > m :  rendre +1
        Résultat <- +1
    fin si
postcondition
    n < m => Résultat = -1
    n = m => Résultat =  0
    n > m => Résultat = +1
fin comparer
```

La construction :

```
si <expression 1> alors
    <traitements 1>
sinon si <expression 2> alors
    <traitements 2>
sinon si < ... > alors
    < ... >
sinon
    <traitements n>
fin si
```

permet de réaliser les traitements associés à différentes expressions booléennes `<expression i>`. Lorsque `<expression i>`, i=1, 2, ..., n-1, prend la valeur VRAI, le bloc des instructions `<traitements i>` est exécuté puis le flot de contrôle passe à l'instruction qui suit **fin si**. Si aucun des prédicats `<expression i>` ne prend la valeur VRAI, ce sont les instructions du bloc `<traitements n>` de l'alternant **sinon** qui sont exécutées.

© ENI Editions - All rigths reserved

© ENI Editions - All rigths reserved

Exercices résolus

Exercice résolu 1 : Tri de trois comparables

*On donne trois données de type **T** qui dérivent de **COMPARABLE**.*

Écrire l'algorithme qui classe ces trois données en ordre croissant.

Solution

Les spécifications sont simples et ressemblent à celles du tri de deux éléments.

Algorithme 10 : Tri de trois comparables - Spécification

```
Algorithme classer3
    # Classe a, b et c en ordre croissant.
Entrée
    a, b, c : T -> COMPARABLE
précondition
    aucune
postcondition
    a ≤ b ≤ c
fin classer3
```

La réalisation est cependant moins évidente que celle du classement de deux éléments. Commençons par remettre en ordre a et b :

```
si
    a > b
alors
    # a > b ; c est à placer.
    échanger(a, b) # a ≤ b ; c à placer.
    Placer c par rapport à a et b
sinon
    # a ≤ b ; c à placer.
    Placer c par rapport à a et b
fin si
```

Ainsi, a et b étant remis en ordre, il reste à placer c par rapport à ces deux valeurs. L'opération décrite par la phrase « Placer c par rapport à a et b » est donc indépendante de la valeur que prend l'expression booléenne a > b, condition du premier test réalisé sur ces valeurs. Que l'on ait a > b ou a ≤ b, il faudra réaliser « Placer c par rapport à a et b ». Ceci est révélé par le fait que cette opération de placement doit être réalisée dans les deux alternants. Par conséquent, cette opération de placement ne dépend pas de la valeur de l'expression booléenne a < b et peut être sortie de l'alternative. Cela revient à factoriser l'opération de placement de c par rapport aux valeurs de a et b. On obtient alors :

```
si
    a > b
alors
    # a > b ; c est à placer.
    échanger(a, b) # a ≤ b ; c est à placer.
sinon
    # a ≤ b ; c est à placer ; a et b sont en ordre.
fin si
Placer c par rapport à a et b.
```

Il reste donc à réaliser l'opération « Placer c par rapport à a et b » lorsque a ≤ b. Cette opération s'écrit :

```
# a ≤ b : placer c.
si
    b > c
alors
    # a ≤ b et c < b.
```

© ENI Editions - All rigths reserved

```
    échanger(b, c) # a ≤ c, b < c.
    Vérifier que le placement de c n'a pas modifié l'ordre de a et b
sinon
    # a ≤ b ≤ c
    rien
fin si
```

L'alternant **alors** place `b` et `c` en bon ordre puis s'assure qu'après l'échange des valeurs de `b` et `c`, `a` et la nouvelle valeur de `b` sont en ordre. L'alternant **sinon** est vide puisqu'alors, les trois données sont en ordre. Il serait possible de rédiger une version définitive de l'algorithme, mais on peut remarquer que placer l'opération « `Vérifier que le placement de c n'a pas modifié l'ordre de a et b` » dans l'alternant **sinon** ne change rien puisque les données sont en ordre dans cet alternant. En réalisant cet aménagement inutile, on impose des traitements sans effet, mais l'opération apparaît alors dans les deux alternants. Alors et comme précédemment pour l'opération « `Placer c par rapport à a et b` », cette opération peut être factorisée :

```
# a ≤ b : placer c.
si
    b > c
alors
    # a ≤ b et c < b.
    échanger(b, c) # a ≤ c, b ≤ c.
sinon
    # a ≤ b ≤ c
    rien
fin si
Vérifier que le placement de c n'a pas modifié l'ordre de a et b
```

Enfin, cette dernière opération est réalisée par :

```
si
    a > b
alors
    # b < a et a ≤ c et b ≤ c.
    échanger(a, b) # a ≤ b ≤ c.
sinon
    # a ≤ b ≤ c.
    rien
fin si
```

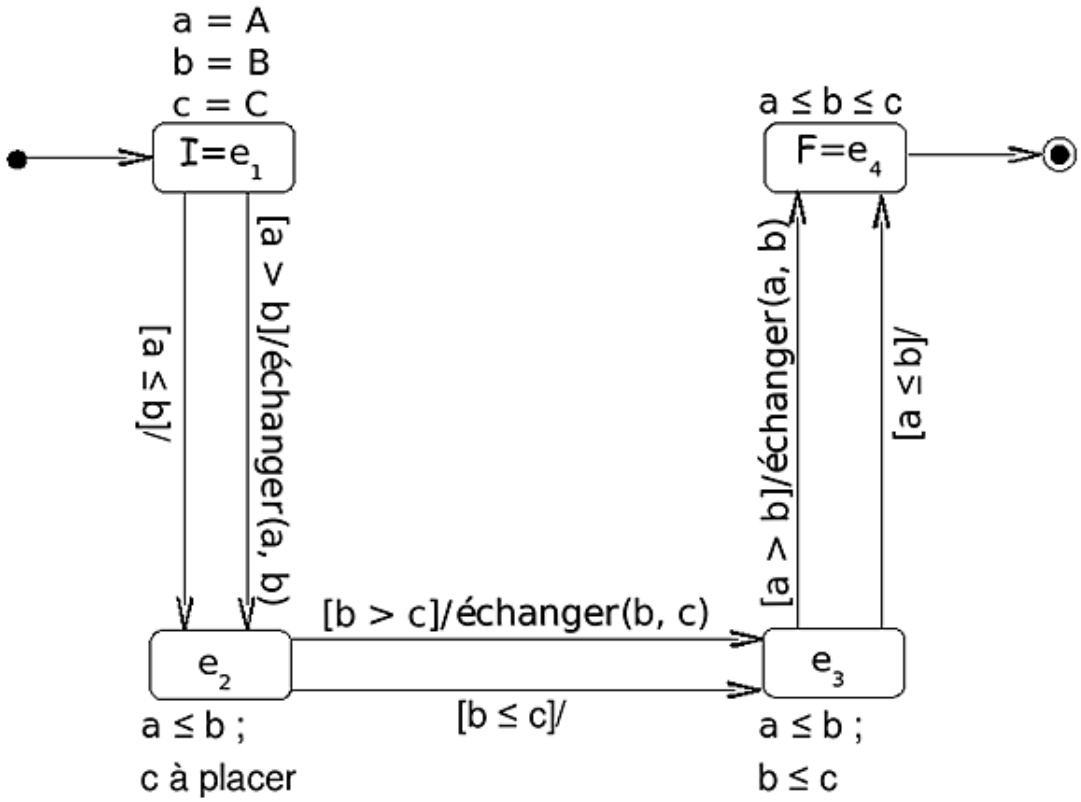
Finalement, l'algorithme définitif est le suivant :

Algorithme 11 : Tri de trois comparables - version 2.0

```
Algorithme classer3
    # Classe a, b et c en ordre croissant.
Entrée
    a, b, c : T -> COMPARABLE
précondition
    aucune
réalisation
    si a > b alors échanger(a, b) fin si # a ≤ b, c reste à placer.
    si b > c alors échanger(b, c) fin si # a ≤ c, b ≤ c.
    si a > b alors échanger(a, b) fin si # a ≤ b ≤ c.
postcondition
    a ≤ b ≤ c
fin classer3
```

Insistons encore sur un point fondamental qui est la thèse de ce livre. C'est le détail des états atteints après chaque instruction qui permet de déterminer les actions à entreprendre et de s'assurer de la correction de l'algorithme. Ce détail est explicité par des commentaires qui décrivent les états par des prédicats sur les variables qui les capturent. Nous procéderons autrement dans les chapitres suivants. Le diagramme de la figure ci-dessous représente ces états et les transitions qui permettent de les parcourir.

© ENI Editions - All rigths reserved



Dans cette version, certains tests sont inutiles pour certaines configurations de valeurs des variables a, b et c. Cependant, on gagne beaucoup en lisibilité ce que l'on perdra éventuellement en efficacité, plus tard, à l'exécution de l'implémentation. C'est souvent le cas en algorithmique et en programmation : il faut faire des choix qui sont des compromis entre lisibilité, efficacité, coût en ressources... Quoi qu'il en soit, le premier objectif de l'analyse algorithmique reste la correction. Il sera bien temps de programmer efficacement lorsqu'une solution correcte aura été obtenue.

Exercice résolu 2 : Reconnaître une année bissextile

Définir un algorithme qui reconnaît si un millésime est celui d'une année bissextile.

Solution

Reconnaître une année bissextile c'est, par exemple, produire le résultat VRAI si le millésime donné est celui d'une année bissextile, ou FAUX sinon. Il s'agit donc d'écrire un prédicat **estBissextile** qui rend VRAI ou FAUX selon que le millésime, qu'il reçoit en paramètre formel, est celui d'une année bissextile ou non ; mais il faut préciser davantage.

Ce prédicat reçoit en entrée un millésime. *Quel est le domaine des valeurs de ce nombre ?* C'est évidemment un nombre entier naturel, mais nous savons que le calendrier que nous connaissons date du 16ème siècle. Considérons, cependant, que parler d'une année bissextile a un sens depuis l'année 1500 et pour tout millésime jusqu'en 2100. Avant, cela n'a pas de sens. Après, il pourrait y avoir des perturbations dans la succession des années bissextiles dues à la rotation de la Terre autour du Soleil.

On a déjà vu au chapitre précédent comment reconnaître une année bissextile. C'est une année dont le millésime est divisible par 4, « à quelque chose près ». Les limites de siècle ne sont pas bissextiles, sauf celles dont le millésime est divisible par 400, comme 2000 par exemple. Ainsi, 1984 et 2000 sont des années bissextiles, alors que 1983, 2005 et 2100 n'en sont pas. Le calcul consiste donc à déterminer d'abord si une année est une limite de siècle. C'est le cas si les deux derniers chiffres du millésime sont nuls. On obtient le nombre constitué des deux derniers chiffres, en calculant le reste de la division du millésime par 100. Ainsi, par exemple, **reste**(1983, 100) = 83. On obtient donc :

```
si
    reste(millésime, 100) = 0
alors
    millésime est une frontière de siècle
sinon
    millésime est une année « normale »
fin si
```

Lorsque millésime est celui d'une frontière de siècle, on calcule le nombre formé des deux premiers chiffres. C'est le

© ENI Editions - All rigths reserved

quotient de millésime par 100. Ce millésime est celui d'une année bissextile lorsque le quotient obtenu est divisible par 4, c'est-à-dire lorsque le reste de la division de ce nombre par 4 est nul. Ainsi :

```
reste(quotient(1900, 100), 4) = 3 # => 1900 n'est pas bissextile.
reste(quotient(2000, 100), 4) = 0 # => 2000 est bissextile.
```

On obtient donc :

```
si
    reste(millésime, 100) = 0
alors
    # Millésime est une frontière de siècle.
    # Est-ce une année bissextile ?
    Résultat <- (reste(quotient(millésime, 100) , 4) = 0 )
sinon
    millésime est une année « normale »
fin si
```

On calcule bien un résultat booléen puisque l'expression à droite du symbole d'affectation est une expression booléenne.

Une année « normale », c'est-à-dire qui n'est pas une limite de siècle, est bissextile lorsque le millésime est un multiple de 4. Ainsi, par exemple, 2004 est une année bissextile. On reconnaît une telle année parce que le reste de la division par 4 du millésime est nul : **reste**(2004, 4) = 0, ce qui donne :

```
sinon
    # Millésime est une année « normale ».
    Résultat <- (reste(millésime, 4) = 0)
fin si
```

Il reste à exprimer la postcondition pour dire ce que fait l'algorithme. On a :

```
postcondition
    Résultat =
    (
        (
        # Millésime est une frontière de siècle.
            reste(millésime, 100) = 0
        et alors
            reste(quotient(millésime, 100), 4) = 0
        )
    ou sinon
        (
        # Millésime est une année « normale ».
        reste(millésime, 4) = 0
        )
    )
```

Cette postcondition exprime un prédicat sur le résultat rendu par l'algorithme. L'expression **ou sinon** évalue le membre de droite lorsque le membre de gauche prend la valeur FAUX. L'expression **et alors** évalue le membre de droite lorsque le membre de gauche prend la valeur VRAI. Ainsi, **Résultat** est égal à VRAI lorsque le millésime est une frontière de siècle divisible par 400 ou sinon lorsqu'il est divisible par 4.

Il reste à rassembler tout cela pour obtenir l'algorithme complet du prédicat.

Algorithme 12 : Reconnaître une année bissextile - Version 1.0

```
Algorithme estBissextile
    # Millésime est-il le millésime d'une année bissextile ?
Entrée
    millésime : ENTIER
Résultat  : BOOLÉEN
précondition
    1500 ≤ millésime ≤ 2100
variable
    année  : ENTIER
    siècle : ENTIER
```


© ENI Editions - All rigths reserved

```
réalisation
    année <- reste(millésime, 100)
    si
        année = 0
    alors
        # Millésime est une frontière de siècle.
        siècle   <- quotient(millésime, 100)
        Résultat <- (reste(siècle, 4) = 0)
    sinon
        # Millésime est une année « normale ».
        Résultat <- (reste(millésime, 4) = 0)
    fin si
postcondition
    Résultat =
    (
        (
        # Millésime est une frontière de siècle.
            reste(millésime, 100) = 0
        et alors
            reste(quotient(millésime, 100), 4) = 0
        )
      ou sinon
        (
        # Millésime est une année « normale ».
        reste(millésime, 4) = 0
        )
    )
fin estBissextile
```

Deux nouvelles variables, `année` et `siècle`, sont utilisées ici. Elles sont initialisées par les résultats de calculs intermédiaires. La première reçoit pour valeur le nombre formé des deux derniers chiffres du millésime. La seconde reçoit le nombre formé des deux premiers chiffres. Ces variables supplémentaires ne sont pas strictement indispensables, comme l'a montré l'analyse du problème. Elles permettent d'écrire des instructions plus simples pour exprimer la réalisation de l'algorithme et le rendent ainsi plus lisible. Cependant, il n'est pas possible de les utiliser dans l'expression de la postcondition. En effet, nous avons déjà dit que tout ce qui n'apparaît pas dans la signature de l'algorithme est local au module. Or, la précondition et la postcondition font partie, avec la signature, de la *documentation publique* du module. Ce sont donc des éléments publiques, communiqués à tous. Tout le reste est privé, local au module et ne peut pas être utilisé dans la partie publique. Ce point apparaît plus clairement quand on se restreint à la partie publique. Elle est donnée ci-dessous.

*Algorithme 13 : Spécifications publiques de **estBissextile***

```
Algorithme estBissextile
    # Millésime est-il le millésime d'une année bissextile ?
Entrée
    millésime : ENTIER
Résultat  : BOOLÉEN
précondition
    1500 ≤ millésime ≤ 2100
postcondition
    Résultat =
    (
        (
        # Millésime est une frontière de siècle.
            reste(millésime, 100) = 0
        et alors
            reste(quotient(millésime, 100), 4) = 0
        )
      ou sinon
        (
        # Millésime est une année « normale ».
        reste(millésime, 4) = 0
        )
    )
fin estBissextile
```

© ENI Editions - All rigths reserved

Exprimer la postcondition en utilisant les variables intermédiaires *année* et *siècle* introduirait des éléments privés au module dans la partie publique, ce qui rendrait cette partie publique incompréhensible sans la connaissance des valeurs de ces deux variables.

Exercice résolu 3 : Signe d'un produit ou d'une somme

1. Écrire un algorithme qui calcule le signe du produit de deux nombres sans calculer le produit.

2. Même question pour la somme de deux nombres.

Solution

Nous avons déjà étudié le signe d'un nombre. Le problème est simple quand on peut calculer le produit ou la somme :

```
# Signe d'un produit.
Résultat <- signe(a x b)
```

et de même pour la somme. Mais l'énoncé est clair : « [...] sans calculer le produit. » ni la somme d'ailleurs.

Pour le produit, la solution reste simple malgré cette contrainte. Il suffit de multiplier les signes des deux nombres :

```
# Signe d'un produit.
Résultat <- signe(a) x signe(b)
```

Lorsque les deux nombres sont non nuls, mais de même signe, la fonction **signe** rend le même nombre pour les deux, soit -1, soit +1, dont le produit vaut +1, indiquant ainsi un produit positif. Lorsque l'un des deux nombres est nul, la fonction **signe** rend 0 et le produit des signes est nul, indiquant que le produit des nombres est nul. Enfin, lorsque les deux nombres sont de signes contraires, on obtient un résultat égal à -1.

Sans la possibilité de calculer le produit des deux nombres, il faut discriminer les différents cas. C'est ce que fait l'algorithme qui suit.

Algorithme 14 : Signe d'un produit - Version 1.0

```
Algorithme signeProduit
    # Le signe du produit de a par b.
Entrée
    a, b : RÉEL
Résultat : ENTIER
précondition
    aucune
réalisation
    si
        (a < 0 et b < 0)
      ou
        (a > 0 et b > 0)
    alors
        # Deux nombres de même signe => produit positif.
        Résultat <- +1
    sinon si
        a = 0 ou b = 0
    alors
        # Au moins un nombre nul => produit nul.
        Résultat <- 0
    sinon
        # Deux nombres de signes contraires => produit négatif.
        Résultat <- -1
    fin si
postcondition
    Résultat = signe(a) x signe(b)
fin signeProduit
```

Remarquez d'abord l'expression de la postcondition. Elle dit le résultat calculé et rendu par l'algorithme, mais il est clair ici qu'elle ne dit pas comment ce résultat est obtenu. D'ailleurs, la réalisation montre cette fois clairement que le résultat n'est pas obtenu par le produit des signes des deux nombres. Encore une fois, la spécification exprime ce que fait l'algorithme et non comment il le fait.

© ENI Editions - All rigths reserved

Il faut encore prouver que la réalisation de cet algorithme fait bien ce qu'annonce la spécification. Montrons, par exemple, que dans le bloc **sinon**, les deux nombres sont bien de signes contraires, comme l'indique le commentaire d'en-tête du bloc.

Lorsque les instructions de ce bloc sont exécutées, c'est que les prédicats qui précèdent ont tous été évalués à FAUX d'après la définition de la construction algorithmique utilisée. On a donc :

```
non [ (a < 0 et b < 0) ou (a > 0 et b > 0)] et non [a = 0 ou b = 0]
```

Cette expression booléenne est équivalente à :

```
non (a < 0 et b < 0) et non (a > 0 et b > 0) et non (a = 0 ou b = 0)
```

ou encore à :

```
(a ≥ 0 ou b ≥ 0) et (a ≤ 0 ou b ≤ 0) et (a ≠ 0 et b ≠ 0)
```

D'où, finalement,

```
(a < 0 et b > 0) ou (a > 0 et b < 0)
```

ce qui établit bien que a et b sont de signes contraires.

Le signe d'une somme de deux nombres est le signe du nombre qui a la plus grande valeur absolue. Lorsque les valeurs absolues sont égales, la somme est nulle si les nombres ont des signes différents, comme -15 et +15 par exemple. Sinon, c'est le signe commun. On obtient l'algorithme suivant :

Algorithme 15 : Signe d'une somme - Version 1.0

```
Algorithme signeSomme
    # Le signe de la somme de a avec b.
Entrée
    a, b : RÉEL
Résultat : ENTIER
précondition
    aucune
réalisation
    si
        abs(a) > abs(b)
    alors
        Résultat <- signe(a)
    sinon si
        abs(a) < abs(b)
    alors
        Résultat <- signe(b)
    sinon si
        signe(a) = - signe(b)
    alors
        Résultat <- 0
    sinon
        Résultat <- signe(a)
    fin si
postcondition
    Résultat = signe(a + b)
fin signeSomme
```

Cet algorithme fait appel aux services du sous-algorithme **abs**. C'est une fonction qui rend la valeur absolue de son paramètre. Cette fonction doit évidemment être définie pour que la fonction **signeSomme** ait un sens. Ce sera fait plus bas. Pour terminer avec **signeSomme**, montrons que lors de l'exécution du second bloc **sinon si**, les valeurs absolues des deux nombres sont égales. On a :

```
non [abs(a) > abs(b)] et non [abs(a) < abs(b)]
```

Ce qui donne :

```
[abs(a) ≤ abs(b)] et [abs(a)≥ abs(b)] <=> abs(a) = abs(b)
```

© ENI Editions - All rigths reserved

a et b ont bien la même valeur absolue dans ce bloc.

Il reste à définir **abs**. C'est fait par l'algorithme suivant, qui exprime directement la définition mathématique de la valeur absolue d'un nombre : c'est le nombre s'il est positif ou son opposé, sinon.

Algorithme 16 : Valeur absolue d'un nombre - Version 1.0

```
Algorithme abs
    # La valeur absolue de a.
Entrée
    a : RÉEL
Résultat : RÉEL
précondition
    aucune
réalisation
    si a ≥ 0 alors Résultat <- a sinon Résultat <- -a fin si
postcondition
    a ≥ 0 => Résultat = a
    a < 0 => Résultat = -a
fin abs
```

Remarquez que la postcondition est une fois de plus une conjonction de clauses. Il est important de noter que les deux conditions doivent prendre la valeur VRAI lorsque l'algorithme se termine.

Cette solution paraphrase la définition mathématique de la valeur absolue. Elle est donc correcte et la façon dont elle obtient la valeur absolue est claire. Ce n'est pas la seule façon de faire. L'exercice suivant propose d'écrire deux nouvelles versions de cette fonction.

Exercice 1 : Valeur absolue d'un nombre

Remarquez d'abord que la valeur absolue d'un nombre se déduit de la valeur de son signe.

*1. Utiliser la valeur rendue par la fonction **signe** pour déterminer la valeur absolue d'un nombre.*

La valeur du signe et la valeur du nombre permettent aussi de calculer sa valeur absolue.

*2. Utiliser la valeur rendue par la fonction **signe** et la valeur du nombre pour déterminer sa valeur absolue.*

Nous avons vu, au chapitre Programmes directs, la fonction **successeur** pour une couleur dans le problème du changement de couleur d'un feu qui règle la circulation d'un carrefour. Cette fonction a été spécifiée, mais n'a pas encore été définie. Nous disposons à présent de tous les outils nécessaires pour en donner une première version pour sa réalisation.

*Algorithme 17 : Opération **successeur** d'une couleur - Version 2.0*

```
Algorithme successeur
    # Le successeur de c dans (rouge ; orange ; vert).
Entrée
    c : COULEUR
Résultat : COULEUR
précondition
    aucune
réalisation
    si
        c = rouge
    alors
        Résultat <- vert
    sinon si
        c = vert
    alors
        Résultat <- orange
    sinon
        Résultat <- rouge
    fin si
postcondition
        c = rouge  => Résultat = vert
```


© ENI Editions - All rigths reserved

```
        c = orange => Résultat = rouge
        c = vert   => Résultat = orange
fin successeur
```

© ENI Editions - All rigths reserved

Exercices

*Exercice 2 : Successeur d'un **JOUR** de la semaine*

*Le type **JOUR** définit par énumération un jour de la semaine. Dans l'exercice qui détermine le jour du 1er Mai d'une année donnée, on a aussi spécifié une fonction successeur pour un jour de la semaine. Il reste à donner une définition de cette fonction.*

*Donner une définition complète de la fonction **successeur** pour un jour de la semaine.*

Nous ne disposons pas encore des outils permettant de donner une définition « élégante » de cette fonction. Ce sera fait plus tard.

Exercice 3 : Nombres, somme et produit

On donne deux nombres quelconques.

Classer ces deux nombres par rapport à leur somme et leur produit.

Ainsi, par exemple, étant donnés a = -15 et b = 6, on obtient a x b < a < a + b < b dont les valeurs sont, dans l'ordre : -90, -15, -9 et 6.

Exercice 4 : Remise

Un commerçant accorde une remise de 5 % pour tout achat d'un montant compris entre 100 et 500 € et 8 % au-delà.

Écrire l'algorithme de calcul du montant de la remise sur un achat donné.

Exercice 5 : Encore une moyenne

Un professeur souhaite écrire un programme qui calcule la moyenne des quatre notes obtenues par ses élèves aux devoirs du mois. Le programme devra, en plus, calculer une appréciation automatique selon la moyenne de l'élève. Il donnera « Élève doué » si la moyenne est supérieure à 15, « Des capacités » si elle est comprise entre 12 et 15 et enfin « Doit se réorienter » si elle est inférieure à 12.

Écrire un algorithme qui prend en entrée les quatre notes d'un élève et qui calcule la moyenne et l'appréciation correspondante.

Le problème précédent peut être résolu en définissant une structure de données qui, pour un élève, regroupe sa moyenne et l'appréciation. C'est un élément de ce type que calcule l'algorithme demandé.

Exercice 6 : Avec la SNCF, c'est possible

La SNCF accorde une réduction aux familles qui se rendent au Futuroscope, en fonction du nombre d'enfants dans la famille. Cette réduction est de 10 % pour 2 enfants, 15 % pour 3 enfants et 18 % pour 4 enfants. À partir de 5 enfants, la réduction est de 18 %, augmentée de 1 % par enfant au-dessus de 4.

Établir l'algorithme qui calcule le montant de la réduction accordée à une famille donnée.

Exercice 7 : Réduction sur les micro-processeurs

La société UNTEL accorde des réductions pour l'achat en masse de ses micro-processeurs. Ces réductions sont fonction du nombre de composants commandés et du client qui les commande.

La réduction consentie est de 10 % si le nombre de composants commandés est compris entre 10000 et 20000, 15 % si ce nombre est compris entre 20001 et 40000 et 20 % pour plus de 40000 composants.

De plus, si le client est COMMAQ, la réduction est réduite de 2 %. Enfin, BEL bénéficie d'une réduction majorée de 1 %.

Établir l'algorithme du calcul du taux de la réduction consentie à un client donné pour une commande donnée.

Exercice 8 : Voyage scolaire

Un professeur envisage d'organiser un voyage scolaire. Le coût du voyage est fonction du nombre d'élèves qui y participe.

Le coût du trajet est de 67,30 € par élève jusqu'à 25 élèves et de 61,00 € au-dessus de 25 élèves. Le coût de la nourriture est de 3,50 € par jour et par élève. L'hébergement, enfin, est de 4,75 € par jour et par élève si le nombre d'élèves est inférieur à 30, 4,00 € si ce nombre est compris entre 31 et 35 et 3,50 € au-delà.

© ENI Editions - All rigths reserved

Établir l'algorithme de calcul du prix de revient par élève et du coût global du voyage en fonction du nombre d'élèves.

Exercice 9 : Prime annuelle

L'entreprise BOURDON attribue en fin d'année une prime annuelle aux chauffeurs routiers qu'elle emploie.

La prime annuelle est entièrement attribuée au chauffeur, à condition qu'il n'ait pas eu d'accident avec une responsabilité supérieure ou égale à 20 % pendant l'année écoulée. Au-delà de 20 % de responsabilité, le chauffeur est considéré par l'entreprise comme responsable de l'accident. Si le chauffeur a été responsable d'un accident, il ne perçoit que la moitié de la prime. Pour deux accidents, il n'en perçoit que le tiers. Pour trois accidents, elle est réduite au quart. Au-delà, la prime est annulée.

Cette prime est la somme d'une prime de distance et d'une prime d'ancienneté.

- *La prime de distance s'élève à un centime par kilomètre parcouru pendant l'année avec un plafond de 400 € ;*
- *la prime d'ancienneté n'est attribuée qu'à partir de quatre années d'ancienneté révolues, soit 200 € dans ce cas. Elle est ensuite augmentée de 20,00 € par année supplémentaire.*

Écrire l'algorithme de calcul de la prime annuelle accordée à chaque chauffeur.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a présenté l'alternative. C'est une construction algorithmique qui permet de choisir un traitement parmi plusieurs, selon les valeurs d'un ensemble de prédicats. Ces valeurs sont obtenues à partir de l'état du système logiciel. Les traitements qui en résultent induisent le plus souvent un changement de cet état. L'alternative se note de différentes façons qui ne sont contraintes que par l'exigence de lisibilité et de correction des algorithmes obtenus.

Les instructions opérationnelles d'un algorithme, comme celles d'un programme informatique d'ailleurs, ne disent rien sur les états intermédiaires du système obtenu à la suite de l'exécution de ces instructions. Expliciter ces états est de première importance pour assurer la correction de l'algorithme. Cependant, l'alternative est une exception dans la mesure où le choix qu'elle prépare résulte de la valeur de prédicats qui, eux, expriment une propriété de l'état dans lequel se trouve le système logiciel.

© ENI Editions - All rigths reserved

Introduction

Les chapitres précédents ont montré que l'algorithmique traite de la transformation de données. Dans ce chapitre, on s'intéresse à la façon de définir et d'organiser les données utilisées par nos algorithmes.

La section Les chaînes de caractères introduit les caractères et les chaînes de caractères. On peut ainsi commencer à étudier les transformations de données textuelles ou, plus généralement, de données non numériques. La section Le tableau définit les tableaux et les premières opérations élémentaires sur ces structures. Tous les langages de programmation des ordinateurs proposent ces types qui sont prédéfinis et directement utilisables. Cependant, ils ne couvrent pas toutes les possibilités de traitement et tous les besoins des programmeurs. On doit donc introduire la possibilité de définir de nouveaux types de données, comme le type **RÉSERVOIR** du chapitre Programmes directs. C'est ce qui est fait en section Définir un nouveau type de données qui étudie comment l'utilisateur peut définir et utiliser ses propres types à partir des types de base.

© ENI Editions - All rigths reserved

Les chaînes de caractères

Cette section étudie le type **CHAÎNE** qui permet de déclarer des données qui ne sont pas numériques. Ainsi, les opérations usuelles sur les nombres ne sont plus du domaine des chaînes de caractères et nous devrons préciser les opérations applicables aux données de ce type.

Une chaîne est composée de caractères. Le **CARACTÈRE** est l'unité élémentaire d'information considérée dans cette section. La première partie introduit les caractères, les notations et les premières opérations applicables. La seconde section compose les caractères pour former les chaînes et commence l'étude des opérations du domaine. Enfin, la dernière section propose une série d'exercices d'application.

1. Les caractères

Un caractère est une donnée textuelle de base. 'A', '4', '!' sont des caractères. Cependant, un caractère n'est pas nécessairement imprimable, c'est-à-dire humainement représentable. Ainsi, par exemple, un caractère particulier destiné à marquer la fin d'un enregistrement dans un fichier, bien que non représentable sur une feuille de papier, n'en est pas moins un caractère comme un autre.

Pour utiliser une variable destinée à recevoir un caractère, un algorithme de ce livre la déclare ainsi :

```
car : CARACTÈRE
```

et lui affecte une valeur comme à toute autre variable :

```
car <- '*'
```

La définition d'une constante de type **CARACTÈRE** utilisera un séparateur pour délimiter le caractère et l'isoler du reste des instructions. Dans l'exemple ci-dessus, une apostrophe a été placée juste avant et juste après la valeur d'initialisation de car, de sorte qu'il n'y ait pas d'ambiguïté. Le but est de distinguer clairement la valeur à attribuer à la variable du contexte dans lequel elle apparaît. Il est possible, de cette façon, d'utiliser un séparateur quelconque, pourvu que ce séparateur apparaisse clairement en tant que tel. Ainsi, il aurait été possible d'écrire indifféremment :

```
car_1, car_2, car_3 : CARACTÈRE
car_1 <- '*'
car_2 <- "*"
car_3 <- /*/
```

Ces instructions initialisent ainsi car_1, car_2 et car_3 avec le caractère *.

Il est possible de déclarer une variable de type **CARACTÈRE** et elle est alors définie : elle a une existence. Cependant, elle ne possède pas encore de valeur. On convient que, dans ce cas, la variable a la valeur NUL. C'est un caractère particulier qui signe l'état d'un caractère non initialisé. Étant donné un caractère car quelconque, il est toujours possible de lui donner la valeur NUL par une affectation :

```
car <- NUL
```

Ce faisant, la valeur qu'il contenait avant cette instruction est perdue.

Les données de type **CARACTÈRE** sont des données **COMPARABLEs**. Autrement dit, **CARACTÈRE** dérive de **COMPARABLE** et il existe une relation d'ordre total définie sur l'ensemble des données de type **CARACTÈRE**. Ainsi, étant donnés deux caractères quelconques c1 et c2, leur comparaison a toujours un sens et un prédicat construit sur leur valeur n'est jamais indéfini dès lors que les caractères c1 et c2 sur lesquels il porte sont définis. Dans la suite, on suppose que l'ordre des caractères est l'ordre alphabétique usuel pour les lettres. Comme un caractère n'est pas nécessairement une lettre, il faut préciser la signification de la comparaison de deux caractères quelconques. C'est fait plus bas dans cette section.

Nous admettrons que tout caractère, humainement représentable ou non, possède un entier unique qui l'identifie. On ne suppose rien sur cet identifiant, sauf qu'il est entier et que tout caractère en possède un. De plus, les identifiants sont attribués en séquence, entre deux valeurs définies par :

```
constante
    CODE_CAR_MIN : ENTIER <- ???
    CODE_CAR_MAX : ENTIER <- ???
```

Les valeurs effectives de ces deux constantes ne nous intéressent pas. Il nous suffit d'admettre que toute valeur entière comprise entre CODE_CAR_MIN et CODE_CAR_MAX est l'identifiant d'un caractère et que tout caractère possède un identifiant entre ces deux valeurs. Il n'y a donc pas de « trou » entre elles. C'est ce qu'exprime le fait que les identifiants de caractères sont attribués en séquence entre ces deux valeurs.

© ENI Editions - All rigths reserved

On émet l'hypothèse que les chiffres '0', '1', ..., '9' ont des identifiants consécutifs. Ce sont des caractères qu'il ne faut pas confondre avec les nombres entiers 0, 1, ..., 9. De même, les caractères alphabétiques majuscules et minuscules ont des identifiants consécutifs dans l'ordre alphabétique. Cependant, les majuscules et les minuscules ne se suivent pas nécessairement. De même, les chiffres et les lettres ne se suivent pas nécessairement. On a donc que l'identifiant de 'B' par exemple est le successeur de l'identifiant de 'A' et celui de 'b' est le successeur de l'identifiant de 'a', mais il n'existe aucune relation connue entre l'identifiant de 'A' et celui de 'a'. On peut pourtant en trouver une simple :

Exercice 1 : Relation entre les identifiants des caractères

Démontrer que la majuscule et la minuscule d'un même caractère possèdent des identifiants dont la différence est la même, quel que soit le caractère.

On accède à la valeur de l'identifiant d'un caractère car quelconque à l'aide d'une requête **code** dont les spécifications sont :

*Algorithme 1 : Spécifications de la fonction **code** d'un caractère*

```
Algorithme code
    # Identifiant numérique de car.
Entrée
    car : CARACTÈRE
Résultat : ENTIER
précondition
    car ≠ NUL
postcondition
    Résultat = (identifiant numérique de car)
    CODE_CAR_MIN ≤ Résultat  ≤ CODE_CAR_MAX
fin code
```

D'autre part, on obtient le caractère associé à un identifiant donné en utilisant la fonction **caractère** dont l'algorithme ci-dessous donne les spécifications.

*Algorithme 2 : Spécifications de la fonction **caractère***

```
Algorithme caractère
    # Le caractère d'identifiant n.
Entrée
    n : ENTIER
Résultat : CARACTÈRE
précondition
    CODE_CAR_MIN ≤ n ≤ CODE_CAR_MAX
postcondition
    code(Résultat) = n
fin caractère
```

Un caractère n'est donc pas nécessairement un caractère alphabétique. Ainsi, '!', ';', '5'... sont aussi des caractères. Considérons alors les valeurs numériques entières qui s'écrivent en base dix à l'aide d'un seul caractère : 0, 1, ..., 9. Une variable contenant un nombre est déclarée de type **ENTIER**. Cependant, on peut vouloir traiter la représentation textuelle de cette valeur. Il s'agira alors d'obtenir le **CARACTÈRE** qui la représente. Ainsi, par exemple, étant donné le nombre deux cent seize exprimé en base dix, c'est-à-dire l'objet numérique dont la valeur est représentée par 216, cette représentation en base dix utilise trois caractères : '2', '1' et '6' alors que sa représentation textuelle utilise 15 caractères : 'deux cent seize' dans lesquels il ne faut pas oublier de compter les deux espaces, entre 'deux' et 'cent' pour l'un, entre 'cent' et 'seize' pour l'autre.

Le problème est alors le suivant : *étant donné un nombre entier c compris entre 0 et 9, comment calculer le caractère qui le représente en base dix ?*

Soit **chiffre** une fonction qui prend en entrée un entier positif ou nul, mais strictement inférieur à 10 et qui retourne le caractère qui le représente en base dix. On a ainsi, par exemple :

```
chiffre(5) = '5'
```

Le calcul consiste à retourner le caractère dont l'identifiant est le code de '0' augmenté de c puisque les identifiants des chiffres sont consécutifs par hypothèse :

```
code('9') = code('8') + 1 = code('7') + 2 = ...
```


© ENI Editions - All rigths reserved

*Algorithme 3 : Définition de la fonction **chiffre***

```
Algorithme chiffre
    # Le caractère de code c en base dix.
Entrée
    c : ENTIER
Résultat : CARACTÈRE
précondition
    0 ≤ c ≤ 9
réalisation
    Résultat <- caractère(code('0') + c)
postcondition
    Résultat = caractère(code('0') + c)
fin chiffre
```

Les opérations définies sur les entiers ne sont pas applicables aux caractères qui représentent ces entiers en base dix. Ainsi, par exemple, l'addition n'est pas définie sur les caractères et on ne peut pas écrire '5' + '3' qui n'a pas de sens. Dans ce cas, il faut pouvoir transformer un caractère représentant un chiffre de la base dix en un entier. C'est l'opérateur dual du précédent qui réalise cette conversion. Il est défini par l'algorithme ci-dessous qui utilise encore l'hypothèse que les identifiants des chiffres sont consécutifs et ordonnés selon l'ordre naturel.

*Algorithme 4 : Définition de l'opérateur **nombre***

```
Algorithme nombre
    # L'entier représenté par car en base dix.
Entrée
    car : CARACTÈRE
Résultat : ENTIER
précondition
    car ≠ NUL
    '0' ≤ car ≤ '9'
réalisation
    Résultat <- code(car) - code('0')
postcondition
    Résultat = code(car) - code('0')
fin nombre
```

Remarquez la deuxième clause de la précondition. Elle utilise la propriété des caractères d'être comparables. La comparaison emploie le signe ≤ comme s'il s'agissait de nombres. Cette clause suppose, en outre, que l'ordre défini sur les caractères est le même que sur les nombres, au moins en ce qui concerne les chiffres de '0' à '9'.

Exercice résolu 1 : Transformation d'un nombre en chiffres

On donne un nombre : 435.

Écrire les instructions qui transforment ce nombre en caractères.

Le problème général consiste à transformer un nombre quelconque en la suite des caractères qui représentent ce nombre en base dix. C'est le problème de l'édition d'un nombre qui sera traité complètement aux chapitres Récursivité et Itération. Ici, ce que nous savons ne nous permet que de traiter des cas particuliers.

Solution

Le nombre est 435. Il est fait de trois chiffres. La transformation donnera donc trois caractères. Soient car_1, car_2 et car_3 les trois variables de type **CARACTÈRE** qui recevront les résultats. Comment, étant donné le nombre 435, obtenir la liste de ses chiffres ? Le chiffre des unités, 5, est le reste de la division de 435 par 10 :

```
variable
    car_1, car_2, car_3 : CARACTÈRE
    ch : ENTIER
    n  : ENTIER
réalisation
    n  <- 435
    ch <- reste(n, 10)
```

© ENI Editions - All rigths reserved

Lorsque ce nombre est obtenu, il faut le transformer en caractère :

```
car_3 <- chiffre(ch)
```

Il suffit alors de répéter le même procédé sur 43, c'est-à-dire sur le quotient de 435 par 10 :

```
n <- quotient(n, 10)
```

Appliquer les opérations précédentes sur 43 donnera le chiffre 3, puis le quotient 4, ce qui terminera la conversion demandée. En regroupant les opérations, on obtient l'algorithme suivant :

Algorithme 5 : Transformation de 435 en caractères

```
variable
    car_1, car_2, car_3 : CARACTÈRE
    n  : ENTIER <- 435 # Le nombre à convertir.
    ch : ENTIER        # Un chiffre de n.
réalisation
    # Calcul du chiffre des unités.
    Ch    <- reste(n, 10)
    car_3 <- chiffre(ch)

    # Calcul du chiffre des dizaines.
    n     <- quotient(n, 10)
    ch    <- reste(n, 10)
    car_2 <- chiffre(ch)

    # Calcul du chiffre des centaines.
    n     <- quotient(n, 10)
    car_1 <- chiffre(n)
```

2. Les chaînes de caractères

Une chaîne de caractères est toute suite de caractères, imprimables ou non, représentables dans le système. Ainsi, par exemple la suite de caractères :

Les arbres perdent leurs feuilles en automne.

est une chaîne de caractères, indépendamment de la valeur de vérité de ce qu'elle énonce. De même, « Abracadabra », « A345cde! » ou « #toto!/ » sont des chaînes de caractères. La première est faite de 11 caractères alphabétiques dont 5 sont des 'a' ou 'A', 2 sont des 'b'... La seconde est constituée de 8 caractères et la troisième de 7 caractères.

Quand un algorithme doit utiliser une chaîne de caractères, il est possible de déclarer la variable qui la contiendra ainsi :

```
variable
    phrase : CHAÎNE
```

On lui affectera une valeur comme à toute autre variable, en utilisant le symbole d'affectation :

```
phrase <- 'Il fera beau demain.'
```

Comme **CARACTÈRE**, **CHAÎNE** dérive de **COMPARABLE**. La comparaison naturelle des chaînes de caractères utilise l'ordre alphabétique usuel, étendu pour tenir compte des caractères qui ne sont pas alphabétiques. Bien entendu, il existe d'autres relations d'ordre possibles.

Pour noter une constante de type chaîne de caractères, on énumère les caractères qui la composent en l'encadrant par un caractère particulier, utilisé comme délimiteur de début et de fin de chaîne. Dans l'exemple précédent, c'est l'apostrophe qui a été utilisée pour délimiter la chaîne 'Il fera beau demain.'. On convient d'utiliser n'importe quel caractère comme délimiteur en respectant les deux conventions suivantes :

- le caractère délimiteur est le même avant le premier caractère de la chaîne et après le dernier caractère ;
- il ne doit pas être un caractère de la chaîne.

© ENI Editions - All rigths reserved

On peut ainsi écrire, d'une façon équivalente :

```
phrase <- 'Il fera beau demain.'
phrase <- "Il fera beau demain."
phrase <- (Il fera beau demain.(
```

Le premier exemple utilise l'apostrophe, le second le guillemet double et le troisième la parenthèse ouvrante. L'apostrophe simple ne sera pas utilisée pour délimiter la chaîne "aujourd'hui" pour des raisons évidentes : elle contient elle-même une apostrophe. Bien entendu, ces notations sont des conventions et il est tout à fait légitime d'en changer. Il ne s'agit pas de définir un nouveau langage avec une grammaire rigide et contraignante, mais plutôt de fixer un cadre de travail rigoureux, qui ne laisse pas de place aux ambiguïtés du langage courant.

Lorsqu'une chaîne est déclarée sans avoir été initialisée, elle ne contient aucun caractère. Comme pour un caractère, elle a alors la valeur NUL. Cette valeur ne doit pas être confondue avec la valeur d'une chaîne vide, qui ne contient aucun caractère :

```
CHAÎNE_VIDE = ''
```

On peut toujours affecter à une variable de type **CHAÎNE** la valeur de la chaîne vide :

```
variable
    phrase : CHAÎNE       # phrase contient NUL.
    ...
    phrase <- CHAÎNE_VIDE # phrase est une chaîne sans caractère.
```

Ce faisant, la chaîne vide est copiée dans phrase qui ne contient alors aucun caractère. Ceux qu'elle contenait éventuellement avant cette instruction sont perdus. Il convient de bien distinguer la chaîne vide, qui ne contient aucun caractère, d'une chaîne qui n'a reçu aucune valeur, c'est-à-dire qui n'a pas encore été initialisée. On a ainsi :

```
variable
    ch_1, ch_2 : CHAÎNE
...
ch_2 <- ''
```

Dans cet exemple, ch_2 est une chaîne de caractères vide. Elle a été initialisée par une chaîne constante qui ne contient aucun caractère. La variable ch_1 n'est pas encore initialisée. Par conséquent, elle n'a pas encore de valeur et elle contient donc NUL, qui n'est pas la chaîne vide :

```
NUL ≠ CHAÎNE_VIDE
```

Nous convenons enfin que les caractères d'une chaîne qui n'est pas vide sont numérotés. Habituellement, le premier caractère portera le numéro 1, le second le numéro 2... mais ce n'est pas obligatoire. Il nous arrivera de faire des conventions différentes, par exemple de supposer que les caractères sont numérotés à partir de -15 ou de +10 ou de toute autre valeur entière qu'on voudra. La convention de numérotation des caractères est décidée et précisée lorsque la chaîne est déclarée. Sans autre précision, on suppose que c'est la numérotation usuelle. Mais alors, un algorithme qui reçoit en paramètre une chaîne de caractères doit connaître la convention adoptée. Un prédicat, qui appartient au répertoire des opérations applicables aux chaînes de caractères, permet de vérifier qu'un nombre donné est un numéro de caractère valide pour une chaîne donnée. C'est **index_valide**, dont les spécifications sont données par l'algorithme suivant :

*Algorithme 6 : Spécifications du prédicat **index_valide***

```
Algorithme index_valide
    # i est-il un numéro de caractère valide pour ch ?
Entrée
    i  : ENTIER
    ch : CHAÎNE
Résultat : BOOLÉEN
précondition
    ch ≠ NUL
postcondition
    Résultat = (index_min(ch) ≤ i ≤ index_max(ch))
fin index_valide
```

La chaîne ch doit donc avoir été initialisée, même à CHAÎNE_VIDE.

Un index est correct lorsqu'il est compris entre les numéros du premier et du dernier caractère. Les fonctions

© ENI Editions - All rigths reserved

index_min et **index_max** sont duales l'une de l'autre. La première rend la valeur déclarée pour le numéro du premier caractère de la chaîne qu'elle reçoit en paramètre. La seconde rend de même la valeur déclarée pour le numéro du dernier caractère. Les spécifications de **index_min** sont :

*Algorithme 7 : Spécifications de **index_min***

```
Algorithme index_min
    # Le numéro du premier caractère de ch.
Entrée
    ch : CHAÎNE
Résultat : ENTIER
précondition
    ch ≠ NUL
postcondition
    ch = CHAÎNE_VIDE => Résultat = 0
    ch ≠ CHAÎNE_VIDE => Résultat =
    (le numéro du premier caractère de ch)
fin index_min
```

Ainsi, le numéro du premier caractère d'une chaîne vide est 0. Ce qui distingue une chaîne vide d'une chaîne non vide dont on a décidé de numéroter les caractères à partir de 0 est que, pour une chaîne vide, le dernier caractère est aussi de numéro 0 :

```
index_min(CHAÎNE_VIDE) = index_max(CHAÎNE_VIDE) = 0
```

De plus, il est toujours possible de comparer une chaîne de caractères ch à CHAÎNE_VIDE pour déterminer si elle contient ou non des caractères.

La spécification de **index_max** est identique à celle de **index_min**. Elle est laissée en exercice.

*Exercice 2 : Spécification de **index_max***

*Écrire les spécifications de la fonction **index_max**.*

Pour préciser le numéro du premier caractère d'une chaîne, lorsque ce n'est pas 1, on peut, par exemple, l'indiquer entre crochets dans la déclaration de la variable, comme ceci :

```
variable
    ch : CHAÎNE[5]
```

Ici, ch est une chaîne dont le premier caractère portera le numéro 5, le second le numéro 6... La chaîne n'étant pas encore initialisée, les fonctions **index_min** et **index_max** ne sont pas applicables. Comme d'habitude, il est tout à fait possible d'adopter d'autres conventions.

Étant donnée `ch`, une chaîne de caractères quelconque, il existe un accesseur **item** qui permet de réaliser une copie d'un caractère de ch désigné par son numéro.

*Algorithme 8 : Spécifications de l'accesseur **item***

```
Algorithme item
    # Le caractère dans ch de numéro rang.
Entrée
    ch   : CHAÎNE
    rang : ENTIER
Résultat : CARACTÈRE
précondition
    ch ≠ NUL
    index_valide(ch, rang)
postcondition
    ch = CHAÎNE_VIDE => Résultat = NUL
    ch ≠ CHAÎNE_VIDE => Résultat =
        (le caractère dans ch de numéro rang)
fin item
```

L'accesseur distingue une chaîne vide d'une chaîne non vide. Une chaîne vide n'a aucun caractère, mais rang = 0 est un index valide. Dans ce cas, **item** rend `NUL`.

© ENI Editions - All rigths reserved

Comme d'habitude, les paramètres en entrée ne sont pas modifiés. Autrement dit, la postcondition a pour valeur la conjonction de quatre valeurs booléennes :

```
postcondition
    ch = CHAÎNE_VIDE => Résultat = NUL
    ch ≠ CHAÎNE_VIDE => Résultat =
        (le caractère dans ch de numéro rang)
    ancien(ch)   = ch
    ancien(rang) = rang
```

ce qui exprime bien que **item** rend une copie du caractère de la chaîne sans la modifier.

Arrivé à ce point, il devient possible de résoudre quelques exercices qui font intervenir des chaînes de caractères.

Exercice résolu 2 : Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères est le nombre de caractères qui la composent. Ainsi, ***longueur****('coucou') = 6 puisque ce mot est composé de 6 caractères.*

Spécifier puis réaliser l'algorithme de la fonction ***longueur****.*

Solution

Il est possible de donner plusieurs solutions à ce problème. Un solution immédiate utilise ce que nous savons déjà : pour toute chaîne de caractères définie, les fonctions **index_min** et **index_max** donnent respectivement les numéros du premier et du dernier caractère. Leur différence, augmentée d'une unité donne alors le nombre de caractères demandé.

Algorithme 9 : Calcul de la longueur d'une chaîne - Version 1.0

```
Algorithme longueur
    # Le nombre de caractères de ch.
Entrée
    ch : CHAÎNE
Résultat : ENTIER
précondition
    ch ≠ NUL
réalisation
    si
        ch = CHAÎNE_VIDE
    alors
        Résultat <- 0
    sinon
        Résultat <- index_max(ch) - index_min(ch) + 1
    fin si
postcondition
    ch = CHAÎNE_VIDE => Résultat = 0
    ch ≠ CHAÎNE_VIDE => Résultat = index_max(ch) - index_min(ch) + 1
fin longueur
```

Une autre solution, plus simple mais moins générale, convient pour les chaînes quand on pose par convention que le numéro du premier caractère est toujours 1. On s'interdit ainsi de numéroter les caractères à partir d'un nombre quelconque et on a toujours **index_min**(ch) = 1 pour toute chaîne ch. La longueur de la chaîne, son nombre de caractères, est alors donné par **index_max**(ch). Ainsi, **index_max** et **longueur** donnent le même résultat : elles sont synonymes. On peut alors adopter l'une et « oublier » l'autre. Le plus simple est d'adopter **longueur** et de laisser les fonctions sur les index.

L'exercice suivant est plus difficile et nous n'avons pas encore tous les éléments pour le résoudre. Ce sera fait, pour une première version récursive au chapitre Récursivité et pour une version itérative au chapitre Itération. Cependant, nous pouvons déjà définir le problème et le spécifier partiellement.

On donne une chaîne de caractères ch et un caractère car. *car est-il un caractère de ch ?*

Il s'agit de déterminer, pour une chaîne et un caractère définis, si le caractère compose la chaîne, c'est-à-dire si le caractère est un composant de numéro compris entre **index_min**(ch) et **index_max**(ch). Soit **appartient** le prédicat à réaliser. L'algorithme ci-dessous donne une première version de la solution.

Algorithme 10 : Première version de ***appartient***

© ENI Editions - All rigths reserved

```
Algorithme appartient
    # car est-il un caractère de ch ?
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
Résultat : BOOLÉEN
précondition
    ch  ≠ NUL
    car ≠ NUL
réalisation
    Résultat <- estDans(ch, index_min(ch), index_max(ch), car)
postcondition
    Résultat  = estDans(ch, index_min(ch), index_max(ch), car)
fin appartient
```

En fait, cette fonction utilise les services d'un autre prédicat, **estDans**, qui détermine si le caractère recherché est présent dans la chaîne entre deux caractères de numéros donnés. Ainsi, **estDans** est plus général que **appartient** et il doit être défini pour que la définition de **appartient** ait un sens. Mais voyons d'abord comment utiliser **estDans**.

Soient, par exemple, les déclarations suivantes :

```
variable
    phrase : CHAÎNE    <- 'Il fait beau.'
    car    : CARACTÈRE <- 'a'
```

On définit ainsi une chaîne `phrase` initialisée à 'Il fait beau.' et un caractère `car` initialisé à la lettre 'a'. Les caractères de phrase sont numérotés à partir de 1. Pour déterminer si `phrase` contient un 'a' entre son caractère de numéro 4 et le caractère de numéro 10, on écrit par exemple :

```
...
si
    estDans(phrase, 4, 10, car)
alors
    la chaîne phrase contient car entre
    les caractères de numéro 4 et 10
sinon
    la chaîne phrase ne contient pas car entre
    les caractères de numéro 4 et 10
fin si
...
```

La fonction **appartient** utilise ce prédicat, mais entre le premier et le dernier caractère pour explorer toute la chaîne. Pour répondre à la même question, mais pour toute la chaîne, il est possible d'écrire, par exemple :

```
...
si
    appartient(phrase, car)
alors
    la chaîne phrase contient car
sinon
    la chaîne phrase ne contient pas car
fin si
...
```

Il serait aussi possible d'utiliser **estDans** de la même façon que **appartient** l'utilise.

Il reste à écrire **estDans**. Commençons par les spécifications.

La signature du prédicat est :

```
estDans
  (
    ch : CHAÎNE           # La chaîne à explorer.
    début, fin : ENTIER   # L'intervalle d'exploration.
    car : CARACTÈRE       # Le caractère à chercher.
  ) : BOOLÉEN
# car est-il un caractère de ch de numéro compris
```


© ENI Editions - All rigths reserved

```
entre début et fin ?
```

La précondition impose que les paramètres soient définis et que les index soient valides. Elle est la suivante :

```
précondition
    ch  ≠ NUL
    car ≠ NUL
    index_valide(ch, début)
    index_valide(ch, fin)
```

C'est dans l'expression de la postcondition que se trouve ce qui est nouveau. Ce sera traité dans les chapitres suivants et cet exercice est abandonné là pour ce chapitre. Cependant, il semble important de noter, dès à présent, qu'aucune autre condition n'est imposée aux indices `début` et `fin`. La fonction doit donc rendre un résultat correct, même lorsque `début > fin`, ce qui est contre-intuitif puisque, à la lecture de la précondition, on s'attend à ce que `début ≤ fin`. En fait, la postcondition précisera que, pour `début > fin`, la fonction **estDans** rend FAUX. D'autres spécifications sont évidemment possibles.

Il faut distinguer soigneusement un caractère, qui est l'unité élémentaire d'information sur les chaînes, d'une chaîne de caractères composée d'un caractère unique. La fonction **longueur** par exemple, est applicable à une chaîne de caractères quelconque, donc aussi à une chaîne ne contenant qu'un seul caractère. Cependant, elle n'est pas applicable à une instance de **CARACTÈRE** : elle n'a pas de sens pour un caractère. Il semble donc utile de disposer d'une fonction **chaîne** applicable à un caractère quelconque et qui rend la chaîne réduite à ce seul caractère. Le résultat est donc de longueur 1. Les spécifications de cette fonction de conversion sont données dans l'algorithme suivant.

*Algorithme 11 : Spécifications de la fonction **chaîne***

```
Algorithme chaîne
    # La chaîne dont le seul caractère est car.
Entrée
    car : CARACTÈRE
Résultat : CHAÎNE
précondition
    car ≠ NUL
postcondition
    car = item(Résultat, index_min(Résultat))
    longueur(Résultat) = 1
fin chaîne
```

La première clause de la postcondition exprime que le premier caractère de la chaîne **Résultat** est `car` et la deuxième que le résultat n'a qu'un seul caractère.

3. Exercices d'application sur les chaînes de caractères

Exercice 3 : Addition des caractères

*Nous avons vu que l'addition n'est pas une opération applicable aux caractères. On veut définir une opération **addition** qui prend en entrée deux caractères représentants des entiers en base dix compris entre 0 et 9 et qui rend le caractère égal à la représentation de leur somme. Ainsi, par exemple, **addition**('3', '2') rend '5'.*

*1. Écrire les spécifications de la fonction **addition**.*

2. Écrire sa définition complète et donner des exemples d'utilisation

Notez bien que **addition** opère sur des caractères et calcule un résultat qui est un caractère. Soigner particulièrement la précondition.

Exercice 4 : Successeur d'un caractère

*On définit le type **CHIFFRE** comme un caractère représentant un entier compris entre 0 et 9 en base dix. Une instance de **CHIFFRE** est donc un élément de l'ensemble {'0', '1', ..., '9'}.*

*1. Définir l'algorithme complet de la fonction **successeur** qui opère sur les instances du type **CHIFFRE**.*

*2. Procéder de même pour définir la fonction **successeur** qui opère sur un caractère majuscule de l'alphabet et celle qui opère sur une minuscule.*

© ENI Editions - All rigths reserved

Le problème doit être défini précisément. Il est possible de le traiter avant ou après le problème suivant.

Exercice 5 : Reconnaître un chiffre, une majuscule, une minuscule

*1. Écrire un prédicat **estChiffre** qui reconnaît un chiffre de la base dix.*

*2. Écrire de même les prédicats **estMajuscule** et **estMinuscule** qui reconnaissent, respectivement, un caractère majuscule ou minuscule.*

*3. Écrire le prédicat **estAlphabétique** qui reconnaît si un caractère est un caractère de notre alphabet.*

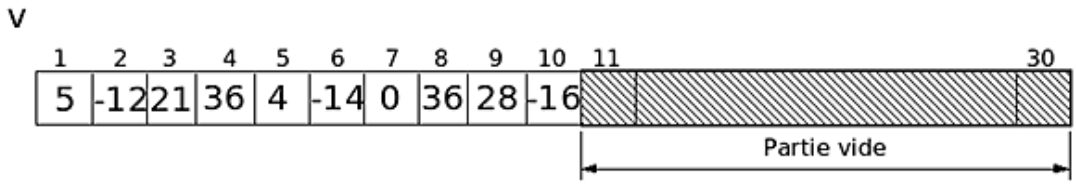
© ENI Editions - All rigths reserved

Le tableau

Un tableau est une structure qui rassemble, sous un même nom, un multi-ensemble de données. Dans la suite, nous supposons que toutes les données enregistrées dans un tableau sont homogènes, c'est-à-dire toutes de même type. On parlera, par exemple, d'un tableau d'**ENTIERs**, d'un tableau de **PERSONNEs**, d'un tableau de **RÉSERVOIRs**... pour exprimer que chacune des données enregistrées dans le tableau est un **ENTIER**, une **PERSONNE** ou un **RÉSERVOIR**. La première section traite des tableaux dont les éléments sont des données, structurées ou non, mais toujours en exemplaire unique. La section suivante étudie les tableaux dont les éléments sont des tableaux. La dernière section propose quelques exercices d'application.

1. Les tableaux simples

Un tableau est constitué d'un ensemble de « cases » numérotées en séquence. Chaque case peut être vide, c'est-à-dire non initialisée, ou contenir une donnée du type dans lequel le tableau a été défini. La figure ci-dessous représente un tableau d'**ENTIERs** de nom v de 30 cases numérotées de 1 à 30.



Les cases numérotées de 1 à 10 ont été initialisées (remplies) avec des entiers. La case numéro 6 contient la valeur -14, la case numéro 1 contient l'entier 5. Les cases dont les numéros vont de 11 à 30 n'ont pas encore reçu de valeur. Elles ne contiennent rien de significatif ou NUL.

Un tableau est défini par sa déclaration. Cette déclaration précise le nom du tableau, le numéro de la première et de la dernière case ainsi que le type de données qu'il contiendra. Ainsi, pour le tableau de la figure ci-dessus, la déclaration est, par exemple :

```
variable
    v : TABLEAU[ENTIER][1,30]
```

Cependant, cette forme n'est pas obligatoire. Ainsi, par exemple, on trouve souvent, chez différents auteurs, la notation suivante :

```
variable
    v : TABLEAU[1 .. 30] d'ENTIER
```

Cette notation imite les déclarations définies en langage PASCAL, par exemple. L'important est encore que la notation utilisée précise bien les différents éléments de la définition du tableau. D'ailleurs, si ce n'était pas si verbeux, il serait tout aussi légitime d'écrire :

```
variable
    v : tableau de 30 cases numérotées de 1 à 30 et recevant des entiers.
```

La déclaration précise le nombre de cases en donnant les numéros de la première et de la dernière case. On peut aussi, par exemple, accepter la convention selon laquelle le numéro de la première case est toujours 1. Alors, le nombre de cases du tableau est le numéro de la dernière case et une déclaration de tableau avec ces conventions se simplifie en :

```
variable
    v : TABLEAU[ENTIER][30]
```

Certaines conventions donnent à la première case du tableau le numéro 0. Dans ce cas, la déclaration qui précède ne donne plus le numéro de la dernière case, mais le nombre de cases.

Nous admettons qu'un tableau est une structure de données statique : après déclaration, le tableau ne peut plus être redimensionné pendant « l'exécution » de l'algorithme. Il est possible, là encore, de faire d'autres conventions et d'admettre que le nombre de cases d'un tableau peut évoluer, mais cela semble compliquer un premier apprentissage.

Comme pour les chaînes de caractères, il est possible de donner à la première case un numéro entier quelconque. La seule contrainte est que le numéro de la dernière case doit lui être supérieur. La déclaration :

© ENI Editions - All rigths reserved

```
variable
    clients  : TABLEAU[PERSONNE] [0, 5000]
    citernes : TABLEAU[RÉSERVOIR][-2, +2]
```

définit deux tableaux. Le premier, de nom `clients`, est un tableau de 5001 cases, chacune contenant une donnée de type **PERSONNE**. Le second est un tableau appelé `citernes`, fait de 5 cases numérotées de -2 à +2 et contenant, chacune, une donnée de type **RÉSERVOIR**.

Pour accéder au contenu d'une case, il suffit d'indiquer, derrière le nom du tableau, le numéro de la case à laquelle on veut accéder. On a ainsi, pour le tableau v représenté par la figure précédente : `v[3] = 21`, `v[7] = 0`, `v[9] = 28`. Pour initialiser la case numéro 15 avec la valeur 156, on écrit :

```
v[15] <- 156
```

Pour placer dans la case 25 la somme des contenus des cases 5 et 10, on écrit :

```
v[25] <- v[5] + v[10]
```

Le membre de droite de l'instruction accède aux cases v[5] et v[10], réalise une copie des valeurs qu'elles contiennent et les additionne. La somme obtenue est rangée dans la case numéro 25 du tableau. Le comportement est donc le même que celui d'une variable quelconque : la copie réalisée à droite du symbole d'affectation <- ne modifie pas les cases auxquelles accède l'instruction. La case destination du résultat, ici v[25], ne peut contenir qu'une seule valeur et, par conséquent, la valeur qu'elle contenait avant l'instruction est perdue.

Les cases de numéros 11, ..., 14, 16, ..., 24, 26, ..., 30 ne contiennent toujours rien.

Comme pour les chaînes de caractères, le prédicat **index_valide** et les fonctions **index_min** et **index_max** permettent d'écrire les spécifications des algorithmes qui utilisent les tableaux.

Exercice résolu 3 : Initialiser un tableau

Écrire la partie d'un algorithme qui initialise un tableau de nom `neuf` avec les résultats de la table de multiplication par 9.

Solution

Une première solution, naïve, énumère les onze cases du tableau en leur donnant le résultat d'une multiplication. La case de numéro 0 contiendra 9x0, celle de numéro 1 contiendra 9x1... Le client logiciel de l'algorithme à écrire aura préalablement déclaré le tableau :

```
variable
    table : TABLEAU[ENTIER][0, 10]
```

Pour utiliser l'algorithme **initialiser** à définir, le client écrira, par exemple :

```
initialiser(table)
```

Il appelle donc les services de **initialiser** en lui communiquant le tableau à préparer. Il est clair que l'algorithme va modifier le tableau. En particulier pour cet exemple, `table` contient `NUL` dans ses 11 cases avant l'appel à **initialiser**. Il contiendra les résultats de la table de multiplication par 9 lorsque **initialiser** aura réalisé son travail.

Algorithme 12 : Table de multiplication par 9 - Version 0.1

```
Algorithme initialiser
    # Table de multiplication par 9.
Entrée
    neuf : TABLEAU[ENTIER]
précondition
    index_valide(neuf, 0)
    index_valide(neuf, 10)
réalisation
    neuf[0]  <- 9 x  0
    neuf[1]  <- 9 x  1
    ...
    neuf[10] <- 9 x 10
postcondition
    les cases neuf[0 .. 10] sont modifiées et contiennent
    les résultats de la table de multiplication par 9
```


© ENI Editions - All rigths reserved

```
fin initialiser
```

neuf est donc le paramètre formel, le marque-place pour le nom du tableau sur lequel opère l'algorithme. Celui-ci modifie le tableau en l'initialisant avec les résultats de la table de multiplication par 9. L'algorithme est donc une procédure.

Cette solution n'est pas bien écrite. Ainsi, par exemple, la précondition force le tableau à être défini de sorte que ses cases soient numérotées depuis 0 pour la première. Ce n'est pas une bonne pratique. La responsabilité de l'algorithme est seulement de calculer la table de multiplication par 9. Il devrait pouvoir réaliser ces calculs sans imposer une déclaration contrainte de la structure de données qu'il utilise. Nous verrons plus tard comment régler ce point. De même, la réalisation n'est pas ce qui se pratique habituellement, mais les outils logiciels pour définir un algorithme « intelligent » n'ont pas encore été mis en place. Ce sera fait dans les chapitres suivants. Bien entendu, il faut remplacer les points de suspension de la réalisation par les instructions qu'ils cachent. C'est facile ici et ce petit travail est laissé en exercice.

2. Tableaux composés

On désigne par l'expression *tableau composé* un tableau dont les cases contiennent des tableaux. Lorsque chaque case du tableau contient un tableau simple, on parle de *matrice*. La table ci-dessous représente une matrice.

	1	***2***	***3***	***4***	***5***
1	12	-8	14	9	0
2	2	4	113	-6	10
3	26	89	18	9	0

La première ligne de cette représentation porte les numéros des colonnes de la matrice. La première colonne porte les numéros de lignes. Chaque ligne est un tableau simple. Ainsi, la ligne 1 est constituée du tableau dont les éléments sont 12, -8, 14, 9 et 0. Par conséquent, en appelant m cette matrice et v, w et z ses trois lignes, `m[1]` désigne le tableau v, `m[2]` désigne le tableau `w` et `m[3]` désigne le tableau `z`. L'élément 113 est donc `w[3]` et comme `w = m[2]`, on obtient `m[2][3] = 113`.

Il est possible de répéter ce raisonnement pour chacun des éléments de la matrice. Ainsi, pour accéder à un élément quelconque, il suffit de donner, à droite du nom de la matrice, son numéro de ligne suivi de son numéro de colonne :

```
m[1][5]=0 ; m[3][1]=26 ; m[1][3]=14 ; m[3][2]=89
```

Pour déclarer un tableau composé, on déclare un tableau en précisant d'abord les numéros de la première et de la dernière ligne, puis les numéros de la première et de la dernière colonne. Ainsi, la matrice m ci-dessus est déclarée par :

```
variable
    m : TABLEAU[ENTIER][1,3][1,5]
```

ou encore :

```
variable
    m : MATRICE[ENTIER][1,3][1,5]
```

ce qui définit une matrice composée de 3 tableaux simples de 5 composantes chacun, soit une matrice de 3 lignes et de 5 colonnes. Il est possible de déclarer de même un tableau de matrices, une matrice de matrices…

3. Exercices d'application sur les tableaux

Exercice 6 : Table de Pythagore

La table de Pythagore est faite de 10 lignes et de 10 colonnes numérotées de 0 à 9. La case à l'intersection de la ligne i et de la colonne j contient le résultat de l'addition i + j ou de la multiplication i x j.

Déclarer puis remplir la table de Pythagore pour la multiplication.

Comme nous ne disposons pas encore des connaissances nécessaires pour remplir cette table « intelligemment », on peut se contenter de montrer comment remplir quelques cases significatives.

© ENI Editions - All rigths reserved

Exercice 7 : Matrice et moyennes des cases

L'exercice reprend la matrice m définie par le tableau précédent. On veut remplir un tableau simple l qui contient, dans chaque case, la moyenne des éléments d'une ligne de m. Ainsi, l[1], par exemple, contiendra la moyenne des éléments de z = m[1].

1. Écrire les instructions qui définissent et calculent les éléments de l.

2. Même question pour un tableau c contenant les moyennes des éléments des colonnes de m.

© ENI Editions - All rigths reserved

Définir un nouveau type de données

Cette section étudie comment définir et utiliser de nouveaux types de données, à l'aide des types élémentaires de base. Cependant, alors que pour les types de base, comme les **ENTIERs** par exemple, les opérations applicables sont implicitement celles des Mathématiques usuelles, pour ces nouveaux types, il faudra clairement et précisément les définir. C'est ce que nous avons déjà commencé à faire dans le chapitre Programmes directs avec le type **RÉSERVOIR**. Cette section propose d'étudier cette question d'une façon plus approfondie.

1. Définir un type de données

Un *type* de données définit le domaine des valeurs que peuvent prendre les données qui sont les instances de ce type, mais aussi les opérations applicables à ces instances. Ainsi, on peut déterminer la quantité de liquide pour **remplir** un **RÉSERVOIR** donné, mais remplir un **ENTIER** n'a pas de sens, de même que additionner deux réservoirs par exemple. **remplir** est une opération applicable à une instance de **RÉSERVOIR** mais pas à une instance d'**ENTIER**.

Un type est défini à partir des *types de base*, c'est-à-dire des types intuitivement évidents et qui font partie de notre quotidien : les nombres entiers, les nombres réels, les booléens... Les opérations applicables aux instances de ces types de base sont habituellement les opérations mathématiques du domaine : additionner, soustraire, multiplier... deux nombres, calculer a **et** b, a **ou** b, pour deux booléens a et b, calculer l'intersection, la réunion de deux ensembles... On compose alors des données de ces types de base pour définir un nouveau type particulier. C'est ce qui a déjà été fait, par exemple, pour définir **RÉSERVOIR** au chapitre Programmes directs. Les opérations applicables ont été définies par des algorithmes complémentaires qui opèrent sur des données du type. Les données qui composent un nouveau type sont les *attributs* du type. Ainsi, un **RÉSERVOIR** possède deux attributs réels : sa `capacité` et sa `contenance`.

Dans ce livre, on note la définition d'un nouveau type par la déclaration :

```
type
    <nom du nouveau type>
structure
    ...
invariant
    ...
fin <nom du nouveau type>
```

Le premier bloc donne le nom qui sera utilisé pour désigner le nouveau type. Ce nom est habituellement précisé en majuscules et en casse grasse dans ce livre. C'est ce nom qui sera utilisé pour instancier des variables du type. Le second bloc donne la liste des clauses qui définissent les attributs du nouveau type. Le troisième bloc regroupe un ensemble, éventuellement vide, de clauses constituées de prédicats. La conjonction de ces prédicats reste toujours vraie, pour toute instance du type, dans tous les états du système et à tout instant. L'exemple ci-dessous définit le type **FRACTION**.

Exemple

Une fraction est définie par la donnée de deux attributs entiers dont le second, le dénominateur, n'est pas nul :

```
type
    FRACTION
structure
    numérateur   : ENTIER
    dénominateur : ENTIER
invariant
    dénominateur ≠ 0
fin FRACTION
```

Une fraction de dénominateur nul n'a pas de sens.

*Les opérations applicables sur une instance de **FRACTION** sont les opérations mathématiques usuelles : normaliser, simplifier, additionner...*

Exemple

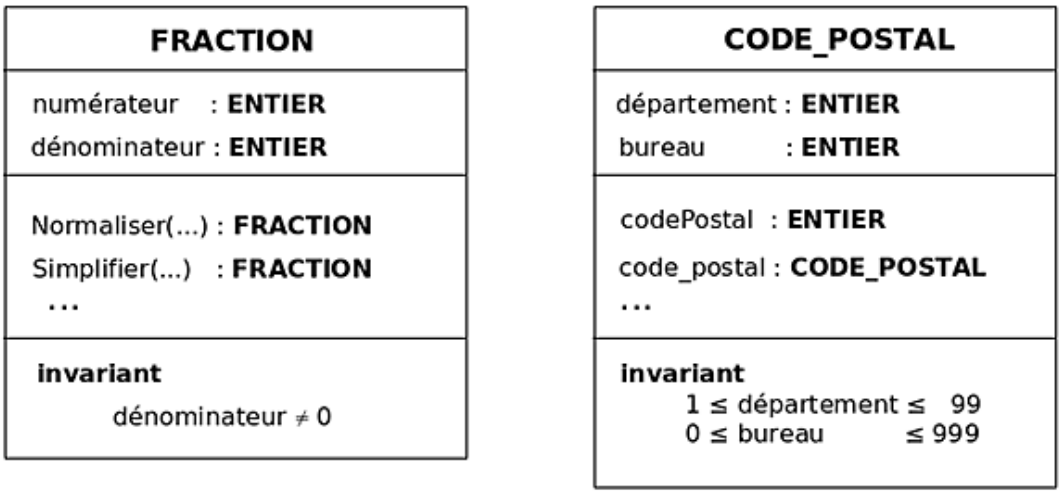
Un code postal est fait d'un attribut entier sur deux chiffres au plus, qui donne le numéro du département Français du bureau de poste qu'il référence, complété d'un attribut entier positif sur trois chiffres, qui est le numéro d'ordre du bureau dans le département.

© ENI Editions - All rigths reserved

```
type
    CODE_POSTAL
structure
    département : ENTIER
    bureau      : ENTIER
invariant
    1 ≤ département ≤  99
    0 ≤ bureau      ≤ 999
fin CODE_POSTAL
```

Une instance de **CODE_POSTAL** n'est pas utilisable immédiatement. En effet, un code postal s'écrit sous la forme d'un nombre entier compris entre 1000 et 99999. Par conséquent, pour rendre utilisable une instance de ce type, on doit fournir une opération **codePostal** qui prend, en entrée, une instance de **CODE_POSTAL** et qui renvoie, en résultat, l'entier qui est le code postal correspondant. Inversement, étant donné un entier, il faut disposer d'une fonction **code_postal** qui prend cet entier en entrée et qui rend l'instance de **CODE_POSTAL** correspondante.

Il est possible de représenter un nouveau type par un diagramme. C'est une boîte rectangulaire qui est divisée en quatre compartiments. Le premier donne le nom du nouveau type. Le second donne la liste des attributs qui le définissent en précisant, pour chacun d'eux, le type de données dont il est une instance. Le troisième compartiment donne les signatures des opérations applicables sur les instances du type. Le dernier compartiment énumère les clauses qui définissent les contraintes sur les valeurs des attributs. La figure ci-dessous représente par un diagramme les types **FRACTION** et **CODE_POSTAL**.



Il est toujours possible de représenter un type par un diagramme incomplet, par exemple avec un bloc d'opérations vide parce qu'elles n'ont pas encore été définies. C'est ce qui est symbolisé par les points de suspension dans les compartiments des opérations sur la figure précédente.

On notera la définition d'une variable qui instancie un type comme on le fait pour un type de base :

```
variable
    v_1, v_2 : <NOM DU TYPE>
```

et l'accès à un attribut particulier d'une variable se note en utilisant l'opérateur *point* (« notation pointée ») pour séparer le nom de la variable du nom de l'attribut auquel on veut accéder :

```
v_1.attribut <- <valeur>
```

Voici, par exemple, la définition de trois fractions f1, f2 et s et quelques instructions qui les utilisent :

```
variable
    f1, f2, s : FRACTION
...
# Initialiser f1 à 4/5.
    f1.numérateur   <- 4 # Le numérateur prend la valeur 4.
    f1.dénominateur <- 5 # Le dénominateur prend la valeur 5.
# Initialiser f2 à 3/2.
    f2.numérateur   <- 3
    f2.dénominateur <- 2
```


© ENI Editions - All rigths reserved

```
# Calculer la somme de f1 et f2.
    s <- addition(f1, f2)
...
```

La section suivante discute la notion d'invariant. Elle peut être ignorée en première lecture. Les débutants en algorithmique peuvent donc passer directement à la résolution des exercices qui suivent cette section.

2. Discussion sur les invariants

Le bloc **invariant** de la définition des types précédents masque une réalité complexe dont cette section présente quelques aspects. Ce sont des considérations théoriques, inspirées de [LIS01] et [MEY00], aux implications difficiles à appréhender lors d'une initiation. Il est possible d'y revenir plus tard.

Reprenons la définition du type **FRACTION** obtenue précédemment :

```
type
    FRACTION
structure
    numérateur   : ENTIER
    dénominateur : ENTIER
invariant
    dénominateur ≠ 0
fin FRACTION
```

La propriété invariante est une propriété de tout objet mathématique qui est une fraction. Pour réaliser une fraction mathématique, une représentation algorithmique, quelle qu'elle soit, doit assurer le maintien de cette propriété. Une fraction de dénominateur nul n'a aucun sens : elle n'est pas définie. Appelons *invariant abstrait* un tel invariant. Il est indissociablement attaché à l'abstraction réalisée par le type algorithmique défini. Cependant, ce n'est pas le seul type d'invariant possible.

Supposons, par exemple, que l'on exige d'une fraction concrète, celle implémentée par l'algorithme, d'avoir un dénominateur positif. Autrement dit, on interdit une fraction comme 5/-4 et on devra toujours la remplacer par -5/4 qui lui est mathématiquement équivalente. L'invariant devient alors :

```
invariant
    dénominateur ≠ 0
    non (dénominateur < 0)
```

ce qui peut aussi s'écrire plus simplement :

```
invariant
    dénominateur > 0
```

Cette fois, l'invariant n'exprime plus une propriété d'une fraction abstraite. C'est une propriété supplémentaire, imposée par la solution algorithmique adoptée. On parle alors d'un *invariant d'implémentation*. Ainsi, on ne pourra plus écrire :

```
f : FRACTION
...
f.numérateur   <-  5
f.dénominateur <- -4
```

Appelons *fraction normalisée* une fraction soumise à cet invariant d'implémentation. Supposons, à présent, qu'une fraction, en plus d'être normalisée, doit aussi être irréductible. Cette fois, l'invariant d'implémentation devient :

```
invariant
    dénominateur > 0
    pgcd(numérateur, dénominateur) = 1
```

dans lequel **pgcd** désigne la fonction qui rend le plus grand diviseur commun de ses paramètres. Ainsi, il ne sera plus possible d'écrire :

```
f : FRACTION
...
f.numérateur   <- 20
f.dénominateur <- 30
```

puisque **pgcd**(20, 30) = 10 ≠ 1.

© ENI Editions - All rigths reserved

3. Exercices d'application sur les types de données

Une personne est une entité dont les attributs sont le prénom, le nom, l'âge et l'adresse. L'adresse complète est d'abord considérée comme une donnée élémentaire.

Exercice résolu 4.1 : Déclaration et utilisation d'un type

*1. Dessiner le diagramme puis définir le type **PERSONNE**.*

*2. Définir (déclarer) deux variables p_1 et p_2 de type **PERSONNE**.*

3. Déclarer une variable `clients` comme un tableau permettant d'enregistrer 200 personnes. Paramétrer le nombre maximum de clients que peut enregistrer le tableau.

4. Faire un schéma représentant ce tableau avec quelques données.

5. Écrire les instructions qui initialisent le 56ème client du tableau comme étant Monsieur Jean DUPONT, qui habite au 36 rue de la Planche à Clous, au MANS, dont le code postal est 72000.

Solution

Le type **PERSONNE** peut être défini ainsi :

```
type
    PERSONNE
structure
    prénom  : CHAÎNE
    nom     : CHAÎNE
    âge     : ENTIER
    adresse : CHAÎNE
fin PERSONNE
```

et deux variables de ce type, ainsi :

```
variable
    p_1, p_2 : PERSONNE
```

ou encore, en dissociant les deux déclarations :

```
variable
    p_1 : PERSONNE
    p_2 : PERSONNE
```

Il est aussi possible d'écrire :

```
p_1 et p_2 sont des PERSONNEs
```

Pour paramétrer le nombre de clients, il suffit d'utiliser une constante. Les déclarations sont, par exemple, les suivantes :

```
constante
    MAX_CLIENTS : ENTIER <- 200
variable
    clients : TABLEAU[PERSONNE][1, MAX_CLIENTS]
```

MAX_CLIENTS est donc une constante manifeste qui remplace, partout où elle apparaît, l'entier 200 dont elle est un substitut.

Le tableau ci-dessous donne un exemple de représentation de la table des clients.

Numéro	*Prénom*	*Nom*	*Âge*	*Adresse*
17	Marcel	MARTIN	27	25 rue des Lilas 72700 ALLONNES
212	Henri	DUPONT	50	57 rue Blanche 72000 LE MANS

© ENI Editions - All rigths reserved

La colonne « numéro » porte un numéro de client et non pas un numéro de case.

Le 56ème client est Monsieur Jean DUPONT, qui habite au 36 rue de la Planche à Clous, au MANS. Pour initialiser la case de numéro 56 du tableau clients, il suffit d'utiliser les instructions suivantes :

```
client[56].prénom  <- 'Jean'
client[56].nom     <- 'DUPONT'
client[56].âge     <- 53
client[56].adresse <- '36 rue de la Planche à Clous 72000 LE MANS'
```

Entre crochets figure le numéro de la case initialisée. Ce numéro de case ne doit pas être confondu avec le numéro de client qui figure dans la première colonne du tableau. Ici, ce numéro de client n'est pas initialisé par les instructions précédentes.

La procédure **écrire** permet d'écrire sur un périphérique donné, c'est-à-dire d'envoyer des données vers ce périphérique. Une telle procédure réalise donc ce qu'il est convenu d'appeler, en informatique, une opération d'entrée/sortie. Ce type d'opération n'est pas, à quelques exceptions près qui seront discutées dans un chapitre ultérieur, de la responsabilité de l'algorithmique. C'est de la programmation et il ne semble pas nécessaire d'utiliser d'autre langage qu'un langage de programmation pour indiquer une opération d'entrée/sortie. Cependant, certains auteurs du domaine veulent à toute force « écrire » et « lire ». Aussi, cédons à la mode pour quelques paragraphes et voyons ce que nous pouvons faire de l'opération d'écriture.

La procédure **écrire** prend en paramètres un nom interne de fichier périphérique, c'est-à-dire un numéro logique qui désigne un périphérique de sortie, et une chaîne de caractères par exemple. Son effet est d'envoyer la chaîne de caractères au périphérique de sortie. On suppose que les numéros de périphérique de l'écran, du clavier et de l'imprimante sont prédéfinis. Ils sont désignés par les constantes ÉCRAN, CLAVIER...

Pour envoyer vers une imprimante la chaîne de caractères 'Il fait beau.', il suffit d'écrire, dans un algorithme :

```
écrire(IMPRIMANTE, 'Il fait beau.')
```

L'hypothèse est que, alors, l'imprimante « correctement connectée » (à l'algorithme ?) écrira, sur la feuille de papier de son magasin, la phrase telle qu'elle se présente en second paramètre.

Exercice résolu 4.2 : Quelques opérations d'entrée/sortie

1. Écrire, à l'écran, les attributs du client enregistré dans la case 67 du tableau.

2. Saisir au clavier les attributs d'une personne et les enregistrer dans les positions réservées au client de la case 186.

Solution

Pour envoyer à l'écran les attributs du client enregistré dans la case numéro 67 du tableau, il est possible, par exemple, d'écrire les instructions :

```
écrire(ÉCRAN, clients[67].prénom)
écrire(ÉCRAN, clients[67].nom)
écrire(ÉCRAN, clients[67].âge)
écrire(ÉCRAN, clients[67].adresse)
```

Mais encore une fois, ce n'est plus de l'algorithmique : c'est de la programmation, la pire qui soit d'ailleurs, puisque ces instructions ne font qu'imiter imparfaitement des instructions de tel ou tel langage. Pourquoi ne pas écrire, dans ce cas par exemple :

```
(void) sprintf("%s\n",clients[67].prenom) ;
```

Voici enfin la saisie interactive des propriétés d'un nouveau client :

```
...
écrire(ÉCRAN, 'Quel est le nom du client ?')
clients[186].nom <- lire(CLAVIER)
écrire(ÉCRAN, 'Quel est son prénom ?')
clients[186].prénom <- lire(CLAVIER)
...
```

On procéderait de même pour saisir l'âge et l'adresse du nouveau client.

Ces instructions distinguent les comportements des opérations d'entrée/sortie et donc la façon d'utiliser **écrire** et **lire**. La première est une procédure alors que l'autre est une fonction. La seconde, **lire**, est une fonction qui rend un résultat : elle « interroge » le système et retourne la réponse obtenue à l'algorithme qui l'utilise. L'opération **écrire** est

© ENI Editions - All rigths reserved

de nature différente. Elle commande au système de modifier son état et, en principe, elle ne renvoie aucun résultat.

*Exercice 8 : Fonctions applicables à une instance de **CODE_POSTAL***

*Écrire les algorithmes des fonctions **codePostal** et **code_postal**.*

*Exercice 9 : Type **CARACTÈRE***

*Faire un diagramme représentant le type **CARACTÈRE** en plaçant les opérations étudiées dans les sections précédentes.*

L'exercice suivant est une variation sur le même thème que l'exercice définissant les réservoirs.

Exercice 10 : Compte en banque et opérations applicables

*On considère le type de données **COMPTE** dont les instances sont les comptes courants des clients d'une banque. On suppose d'abord que la banque n'autorise aucun découvert. Toute opération conduisant à un découvert est illégale et n'est pas acceptée.*

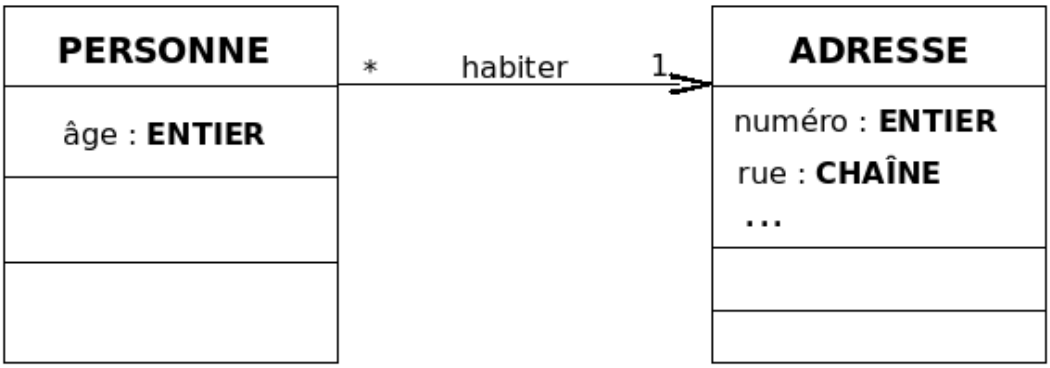
*1. Définir le type **COMPTE** en précisant les attributs et les opérations applicables.*

On suppose ensuite que certains clients ont un découvert autorisé, mais limité à une somme fixée par la banque à l'ouverture du compte. Une opération sur le compte n'est légale que si le solde du compte reste supérieur ou égal à la limite inférieure fixée.

2. Redéfinir le type précédent et les opérations.

On reprend à présent la définition du type **PERSONNE**. Ce type utilise une adresse définie précédemment comme une chaîne de caractères. Il s'agit d'en faire une instance d'un nouveau type de données à définir.

L'adresse d'une personne est un type dont les attributs sont, pour simplifier, un numéro dans une rue, une rue, un code postal et une ville. On suppose qu'une personne donnée ne possède qu'une seule adresse ; soit sa résidence principale. Il existe donc une *association* entre une **PERSONNE** et une **ADRESSE**. Cette association est notée comme sur le diagramme de la figure ci-dessous.



Le diagramme ne porte ni les opérations, ni les clauses d'invariant. L'association est représentée par le trait qui relie les deux entités. Le nom de l'association est indiqué sur le trait. Ici, il est indiqué que toute instance de **PERSONNE** habite à une **ADRESSE**. Le 1 porté au niveau de **ADRESSE** signifie qu'une personne quelconque habite à une adresse unique. L'étoile portée au niveau de l'entité **PERSONNE** indique qu'une adresse est associée à aucune, une ou plusieurs personnes. Ainsi, à une même adresse peuvent habiter plusieurs personnes, mais une adresse peut n'être associée à aucune personne. Ce peut être un logement inoccupé par exemple.

Ce diagramme, évidemment incomplet, indique ainsi que le type **PERSONNE** est défini par :

```
type
    PERSONNE
structure
    âge     : ENTIER
    adresse : ADRESSE
fin PERSONNE
```

Remarquez bien que le nom et le prénom ne figurent pas dans cette définition du type **PERSONNE**. Bien entendu, le type **ADRESSE** doit aussi être défini. Ces définitions sont donc incomplètes, mais elles traduisent le diagramme de la figure ci-dessus.

© ENI Editions - All rigths reserved

Exercice 11 : Déclaration et utilisation d'un type composé

*1. Représenter et définir le type **IDENTITÉ** dont les attributs sont un prénom et un nom.*

*2. Représenter complètement puis définir le type **ADRESSE**.*

*3. Refaire un diagramme complet en mettant en évidence l'association entre **PERSONNE** et **IDENTITÉ**.*

Tenir compte de ce que plusieurs personnes peuvent avoir la même identité, même quand elles n'habitent pas à la même adresse. Il suffit de consulter un annuaire téléphonique de la ville de Paris et de rechercher les Jacques MARTIN pour s'en convaincre.

*4. Redéfinir alors le type **PERSONNE**.*

*Un client est une **PERSONNE** identifiée par un numéro entier strictement positif.*

*5. Définir le type **CLIENT**.*

*6. Déclarer la variable `clients` comme un tableau de **CLIENTs** pouvant enregistrer jusqu'à 1000 clients.*

7. Écrire les instructions d'initialisation de l'adresse de Jean DUPONT.

On veut déterminer le nom et le prénom de tous les clients de nom 'DUPONT'.

*8. Donner la signature et montrer comment utiliser un algorithme **chercherNom** pour résoudre ce problème.*

On doit pouvoir enregistrer moins de 1000 clients dans le tableau.

9. Comment marquer la fin des données valides ?

Des clients sont régulièrement ajoutés ou effacés du tableau.

10. Comment indiquer dans le tableau qu'une case était occupée par un client qui a été effacé ?

Dans les questions suivantes, on demande seulement la signature de l'algorithme résolvant le problème et les instructions qui montrent comment l'utiliser.

11. Rechercher la position du premier client enregistré habitant 'LE MANS'.

12. Éditer (c'est-à-dire écrire à l'écran) le numéro, le nom et le prénom de tous les clients habitant 'LE MANS' et prénommés 'Alain'.

L'exercice suivant propose quelques développements d'algorithmes dans le domaine du graphique. Bien entendu, tout cela reste modeste, mais on peut imaginer que l'activité proposée est en relation directe avec les algorithmes étudiés pour présenter des graphiques et des dessins.

Exercice 12 : Dessins

Un point géométrique est représenté par une structure de données qui permet de définir son abscisse et son ordonnée dans un repère cartésien du plan. Un segment est défini par deux points qui sont ses extrémités.

*1. Définir les types **POINT** et **SEGMENT**.*

Un vecteur géométrique est aussi défini par ses deux composantes, l'une correspondant à son abscisse, l'autre à son ordonnée.

*2. Définir le type **VECTEUR**.*

3. Écrire la procédure qui permet de faire subir à un point donné une translation de vecteur donné.

4. Écrire la fonction qui rend la distance d'un point à l'origine du repère.

5. Généraliser la fonction précédente en écrivant la fonction qui retourne la longueur d'un segment.

Un polygone est défini par les points qui forment ses sommets. On peut, par exemple, définir un polygone à l'aide d'un tableau de points. Ce n'est pas la meilleure façon de définir un polygone, mais cela suffit pour une première approche.

*6. Définir les types **POLYGONE**, **RECTANGLE**, **LOSANGE**, **CARRÉ**, **TRIANGLE**.*

Remarquez qu'un rectangle, un losange et un triangle sont aussi des polygones. Un carré est un losange, mais aussi un rectangle...

On donne un polygone.

7. Écrire la fonction qui retourne son périmètre.

© ENI Editions - All rigths reserved

Notes bibliographiques

De nombreux livres et articles traitent des structures de données. C'est même la préoccupation centrale des ouvrages dont le sujet est l'algorithmique. On peut citer, par exemple, [BM83], [CB84], [CV75]... pour un point de vue strictement algorithmique, ou [CAR97] pour des exemples résolus et programmés dans différents langages, comme JAVA, ADA, C++... Peu d'ouvrages récents, abordables pour une première initiation, s'intéressent aux raisonnements de l'algorithmique. Le livre que vous lisez ne s'intéresse pas aux structures de données, mais bien aux raisonnements qui permettent de construire des algorithmes simples. Les structures élémentaires n'apparaissent ici que comme support, comme prétexte à l'analyse d'algorithmes.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a présenté quelques notions élémentaires sur la définition algorithmique de structures de données. Une structure de données permet de déclarer et d'utiliser des variables dont les valeurs sont des instances du type défini par la structure. Une telle instance est faite d'attributs, qui sont des instances des types de base ou d'autres structures, et d'opérations applicables. Nous avons également étudié plus particulièrement la chaîne de caractères qui permet de définir et d'opérer sur des données textuelles et le tableau qui permet de définir des variables regroupant sous un même nom un multi-ensemble de variables de même type.

© ENI Editions - All rigths reserved

Bibliographie

[BM83] Jean-Claude BOUSSARD, Robert MAHL : *Programmation avancée - Algorithmique et structure de données* ; EYROLLES, 1983.

[CB84] G. CLAVEL, J. BIONDI : *Introduction à la programmation - Tome 2 : Structures de données* ; MASSON, 1984.

[CV75] Jacques COURTIN, Jacques VOIRON : *Introduction à l'algorithmique et aux structures de données* - 2 tomes ; Institut Universitaire de Grenoble, Département d'informatique, 1975.

[CAR97] Christian CARREZ : *Structures de données en Java, C++ et Ada95* ; InterEditions, 1997.

[LIS01] Barbara LISKOV *Program Development in Java* ; ADDISON WESLEY, 2001.

[MEY00] Bertrand MEYER : *Conception et programmation orientées objet* ; EYROLLES, PARIS, 2000.

© ENI Editions - All rigths reserved

Introduction

Ce chapitre est le premier qui introduit la notion de *répétition* dans les traitements des données. Cependant, alors que dans les chapitres précédents, les changements d'états du système s'obtenaient par des instructions qui disaient explicitement les modifications à faire subir aux données, ils s'obtiennent ici en donnant une *définition* de ces changements. En particulier, les traitements n'utilisent pas l'instruction d'affectation, sauf marginalement, pour donner une valeur à la pseudo-variable **Résultat** afin de préciser ce que calcule et retourne la fonction.

La deuxième section introduit la récursivité dans les traitements sur les chaînes de caractères. On montre ce qu'est une définition récursive et on applique ce type de définition à la spécification d'algorithmes sur les chaînes de caractères. La section suivante donne quelques exemples de spécifications ou de traitements récursifs sur les nombres. La quatrième section, enfin, traite d'un problème que l'on rencontre souvent : l'édition d'un nombre où il s'agit de transformer un nombre donné pour obtenir les caractères qui représentent sa valeur dans une base de numération donnée.

© ENI Editions - All rigths reserved

Introduction à la récursivité : les chaînes de caractères

La récursivité est introduite dans cette section comme une méthode générale de spécification algorithmique des algorithmes. Cette phrase est déjà, en soi, une phrase qui énonce une propriété récursive : la spécification d'un algorithme est un algorithme.

La première partie présente la récursivité sur un exemple simple : la définition d'un *mot* du langage. La deuxième partie étudie des exemples de spécifications récursives d'algorithmes de traitement sur les chaînes de caractères. La troisième partie donne la solution complète de quelques exercices instructifs et la dernière, enfin, propose de résoudre divers exercices dont la solution n'est pas développée.

1. Présentation de la récursivité

Considérons un *mot*. Un mot est une chaîne de caractères de longueur finie. Pour simplifier cet exposé, on se restreint aux mots, de la langue ou pas, écrits en minuscules et sans caractère accentué : 'maison', 'poisson', 'voiture' sont des exemples de mots qui nous intéressent. 'École', 'Maison', 'élection' sont exclus, du fait des caractères accentués ou des majuscules qu'ils contiennent.

Comment donner une définition formelle d'un mot ?

Il s'agit de donner une définition qui s'applique à un mot quelconque, sans accent ni majuscule, comme on l'a dit. Il est possible d'en donner la définition suivante :

Définition 1

Un mot est soit un caractère, soit un caractère suivi d'un mot.

Autrement dit, un mot, choisi parmi ceux auxquels on se restreint, est soit une chaîne de caractères réduite à un seul caractère ou sinon, une chaîne de caractères constituée d'un premier caractère suivi d'un autre mot. Ainsi, par exemple, le mot 'maison' est constitué du caractère 'm' suivi du mot 'aison'. Le mot 'aison' est formé de 'a' suivi du mot 'ison'... Il suffit de dire, à présent, ce qu'est un caractère. On peut définir un caractère par une énumération :

Définition 2

Un caractère est un élément de l'ensemble {'a', 'b', ..., 'z'}.

Ce qui est remarquable dans la définition d'un mot, c'est qu'elle utilise *mot* pour le définir. Cette définition n'est pas « circulaire » : étant donnée une chaîne de caractères ch de longueur finie, il est toujours possible de décider, en appliquant cette définition, si ch est un mot ou non.

Exemple

Appliquer cette définition à la chaîne de caractères 'lèvre' pour déterminer si elle est un mot au sens de la définition ci-dessus.

'lèvre' est constituée du caractère 'l' suivi de la chaîne 'èvre'. La chaîne 'èvre' est constituée de 'è' suivi de la chaîne 'vre', mais 'è' n'est pas un caractère d'après la définition 2 donnée par énumération, plus haut. Par conséquent, 'lèvre' n'est pas un mot d'après la définition 1.

Exemple

La chaîne de caractères 'pas' est-elle un mot ?

Elle est constituée de 'p' qui est un caractère selon la définition 2, suivi de la chaîne 'as'. Celle-ci est faite du caractère 'a' suivi de la chaîne 's' et 's' est aussi un caractère selon la définition 2. Par conséquent, 'pas' est un mot.

Remarquez aussi que la chaîne candidate doit avoir au moins un caractère : CHAÎNE_VIDE n'est donc pas un mot. De même, NUL n'est pas un caractère.

Étant donnée cette définition, peut-on écrire un prédicat qui reconnaît si une chaîne donnée est un mot ? C'est possible en « copiant » la définition précédente.

On observe d'abord le premier caractère de la chaîne et, s'il est le seul, on peut conclure :

```
longueur(ch) = 1 => Résultat = estCaractère(premier(ch))
```

La fonction **premier** retourne le premier caractère de la chaîne qu'elle reçoit en paramètre. Elle sera définie plus tard.

© ENI Editions - All rigths reserved

Le prédicat **estCaractère** rend VRAI si et seulement si le paramètre est un caractère autorisé. Ainsi, lorsque le premier caractère de la chaîne, qui est aussi le seul, est un caractère au sens de la définition 2, **estCaractère** rend VRAI et c'est cette valeur que doit prendre **Résultat**.

Lorsque la chaîne est faite de plusieurs caractères, on vérifie que le premier caractère est autorisé puis que le reste de la chaîne est un mot :

```
longueur(ch) > 1 => Résultat =
    (
        estCaractère(premier(ch))
      et alors
        estMot(fin(ch))
    )
```

La fonction **fin** retourne la chaîne, qu'elle reçoit en paramètre, privée de son premier caractère. Elle aussi sera définie précisément plus tard.

La définition 1 dit qu'un caractère autorisé est un mot. Nous ne pouvons pas écrire directement :

```
# DÉBUT DE SOLUTION INCORRECTE !!!
longueur(ch) > 1 => Résultat =
    (
        estMot(premier(ch))
      et alors
        estMot(fin(ch))
    )
# FIN DE SOLUTION INCORRECTE
```

puisque **estMot** attend une chaîne de caractères en paramètre, mais **premier** rend un **CARACTÈRE**. L'utilisation de **estMot** avec le résultat de **premier** comme paramètre est incorrecte car elle ne respecte pas les conditions d'usage de **estMot** précisées dans ses spécifications. Finalement, on obtient l'algorithme suivant :

*Algorithme 1 : Reconnaître un mot : **estMot** - Version 1.0*

```
Algorithme estMot
    # ch est-il un mot ?
Entrée
    ch : CHAÎNE
Résultat : BOOLÉEN
précondition
    ch ≠ NUL
    longueur(ch) ≥ 1
postcondition
    longueur(ch) = 1 => Résultat = estCaractère(premier(ch))
    longueur(ch) > 1 => Résultat =
    (
        estCaractère(premier(ch))
      et alors
        estMot(fin(ch))
    )
fin estMot
```

Il n'y aura pas d'affectation dans la réalisation, c'est-à-dire pas d'ordre à exécuter, sauf peut-être, d'une façon marginale, à cause des conventions de notation, l'affectation d'une valeur à la pseudo-variable **Résultat** pour indiquer le résultat calculé et rendu par la fonction.

La réalisation de l'algorithme est simple : elle ne fait que copier la spécification qui, elle-même, ne fait que traduire d'une façon algorithmique les définitions données plus haut. Cette réalisation est la suivante :

```
réalisation
    si
        longueur(ch) = 1
    alors
        Résultat <- estCaractère(premier(ch))
    sinon
        Résultat <-
            (
                estCaractère(premier(ch))
```


© ENI Editions - All rigths reserved

```
            et alors
              estMot(fin(ch))
          )
    fin si
```

Ainsi, cet algorithme ne dit pas comment calculer le résultat, même dans sa réalisation. Il ne fait qu'en donner une définition. Pour cela, il utilise ses propres services, sur la même chaîne de caractères, mais à partir d'un caractère de numéro supérieur. Une définition qui fait appel à un concept pour le définir, comme la définition 1 définit *mot* en utilisant *mot*, est une *définition récursive*. De même, un algorithme qui appelle ses propres services est un algorithme *récursif*.

Définition 3

On dit d'un algorithme qu'il est simplement récursif lorsqu'il s'appelle lui-même pour réaliser ses calculs.

Il existe des formes plus complexes de récursivité que nous aborderons ultérieurement.

Remarquez comment est construite la postcondition de **estMot**. Elle commence par donner le résultat lorsque la chaîne de caractères est faite d'un caractère unique. La chaîne est un mot, selon la définition 1, si son unique caractère est un caractère défini dans l'énumération de la définition 2. Ce cas de chaîne de longueur 1 est le cas particulier qui permet de rendre immédiatement le résultat. Lorsque la chaîne est faite de plusieurs caractères, le résultat prend la valeur VRAI si le premier caractère est un caractère acceptable, au sens de la définition 2, et si la chaîne, privée de ce premier caractère, est aussi un mot. C'est exactement la définition 1. À chaque essai de **estMot**, la chaîne est amputée de son premier caractère. La longueur des chaînes analysées décroît strictement d'une unité. Par conséquent, comme une chaîne est, par définition, de longueur finie, un nombre fini d'opérations conduit à une chaîne de longueur 1 ce qui termine l'algorithme. Il est même possible de donner un majorant exact du nombre d'opérations au terme desquelles l'algorithme produit son résultat : c'est **longueur**(ch). Ainsi, l'algorithme se termine avec une chaîne de longueur 1 et à tout instant, c'est-à-dire dans chaque état du système logiciel, le nombre d'états qui reste à parcourir est égal, au plus, à la longueur de la chaîne résiduelle.

Résumons : la longueur de la chaîne est un entier fini, qui décroît strictement d'une unité à chaque appel de la fonction et qui est minoré par une longueur égale à 1. Cette analyse prouve que l'algorithme se termine en un nombre fini d'étapes.

Ainsi, l'algorithme converge vers le résultat, mais nous avons appris quelque chose d'essentiel ici : nous savons mesurer le nombre maximum d'états à visiter avant d'obtenir le résultat. C'est, au plus, la longueur de la chaîne résiduelle dans l'état considéré.

Exercice 1 : Nouvelle définition d'un mot

On modifie la définition 1 d'un mot en acceptant CAR_VIDE parmi les caractères autorisés.

1. Redéfinir un caractère.

2. Est-il nécessaire de redéfinir un mot ?

*3. Refaire **estMot**.*

Exercice 2 : Définition d'un nombre en lettres

On veut écrire un prédicat qui reconnaît si une chaîne de caractères est la représentation en base dix d'un nombre entier positif ou nul.

1. Écrire la définition d'un tel nombre.

2. Donner la spécification d'un algorithme qui reconnaît un nombre selon cette définition.

Remarquez qu'un nombre ici est une chaîne de caractères. Il est donc possible de définir un nombre en s'inspirant de la définition 1 : un nombre est un caractère ou un caractère suivi d'un nombre. Il reste alors à préciser la définition d'un caractère : c'est un élément de {'0', '1', ..., '9'}.

2. Quelques exemples de spécifications récursives

Le chapitre Structures élémentaires a commencé l'étude de la définition du prédicat **estDans** qui détermine si un caractère compose une chaîne donnée. Dans cette section, nous étudions une version de la réalisation de cette fonction. Le chapitre suivant étudiera une nouvelle version. La section suivante développe des exemples complets et la dernière section propose des exercices d'application.

La signature de **estDans** était la suivante :

```
Algorithme estDans
```

© ENI Editions - All rigths reserved

```
    # car est-il un caractère qui compose ch entre les caractères
    # de numéros début et fin ?
Entrée
    ch   : CHAÎNE       # La chaîne à explorer.
    début, fin : ENTIER # Les numéros des caractères de l'intervalle
                        # d'exploration.
    car  : CARACTÈRE    # Le caractère à chercher.
Résultat : BOOLÉEN
précondition
    ch  ≠ NUL
    car ≠ NUL
    index_valide(ch, début)
    index_valide(ch, fin)
```

On peut d'abord remarquer que, lorsque `début > fin`, le domaine de recherche est vide et, par conséquent, le prédicat rend FAUX :

```
début > fin => Résultat = FAUX
```

Lorsque l'intervalle d'exploration est réduit à un seul caractère, il suffit de vérifier que ce caractère est égal ou non au caractère cherché. L'intervalle est réduit à un seul caractère lorsque `début = fin`. Pour déterminer si car est ce seul caractère, il suffit de le comparer à une copie obtenue à l'aide de la fonction **item** :

```
si
    car = item(ch, début)
alors
    car est le caractère de numéro début dans ch.
sinon
    car n'est pas le caractère de numéro début dans ch.
fin si
```

Comme le résultat est un booléen, VRAI si et seulement si car est dans ch, ceci se simplifie en :

```
début = fin => Résultat = (car = item(ch, début))
```

Lorsque début < fin, le résultat est VRAI si car est le caractère de numéro début ou sinon, si car est dans le reste de la chaîne, entre les caractères de numéros début+1 et fin.

```
début < fin => Résultat =
    (
        (car = item(ch, début))
      ou sinon
        (estDans(ch, début+1, fin, car))
    )
```

Cette définition utilise le prédicat pour le définir. La postcondition utilise **estDans** pour dire ce qu'il fait : on regarde si car est le premier caractère du domaine à explorer. Si ce n'est pas le cas, on regarde s'il est dans le domaine, mais privé du premier caractère, en utilisant encore **estDans**. À chaque appel, le domaine à explorer est réduit de son premier caractère. Par conséquent, le numéro du premier caractère à observer dans ch augmente en se rapprochant de la valeur de fin. Cette exploration terminera puisque la clause :

```
début = fin => Résultat = (car = item(ch, début))
```

termine l'exploration si car n'est pas trouvé avant. L'expression entre parenthèses :

```
(car = item(ch, début))
```

est une expression booléenne qui prend la valeur VRAI si et seulement si le caractère car est égal au caractère numéro début de ch dont **item** réalise une copie. L'expression :

```
Résultat = ...
```

est aussi une expression booléenne qui rend VRAI si et seulement si la pseudo-variable **Résultat** est égale à la valeur rendue par l'expression précédente.

Pour comprendre le fonctionnement de cette spécification, détaillons l'exemple présenté plus haut, où l'on cherche 'a' dans la phrase 'Il fait beau.', entre les caractères de numéro 4 et 10.

© ENI Editions - All rigths reserved

Premier appel :

```
    estDans('Il fait beau.', 4, 10, 'a')
      début=4 < fin=10 => Résultat =
      (
          ('a' = item('Il fait beau.', 4))
        ou sinon
          (estDans('Il fait beau.', 5, 10, 'a')
      )
```

On a donc :

```
item('Il fait beau.', 4) = 'f' ≠ 'a'.
```

Deuxième appel :

```
    estDans('Il fait beau.', 5, 10, 'a')
      début=5 < fin=10 => Résultat =
      (
          ('a' = item('Il fait beau.', 5))
        ou sinon
          (estDans('Il fait beau.', 6, 10, 'a')
      )
```

On a donc :

```
item('Il fait beau.', 5) ='a'
```

et, par conséquent, le second appel : `estDans('Il fait beau.', 5, 10, 'a')` retourne VRAI. En reportant ce qui vient d'être obtenu dans le premier appel, nous obtenons :

```
début=4 < fin=10 => Résultat = (FAUX ou sinon VRAI)
```

Ainsi, le résultat rendu par l'algorithme est VRAI, signifiant que 'a' a été trouvé dans 'Il fait beau.' entre le caractère de numéro 4 et le caractère de numéro 10.

Cependant, nous n'avons rien dit d'une recherche dans une chaîne vide. *Que se passe-t-il si ch est la chaîne vide ?* Cette question est-elle sans objet, autrement dit, le raisonnement précédent inclut-il ce cas ou doit-on se préoccuper de ce cas marginal ?

Lorsque la chaîne cible de la recherche est vide, on a début = fin = 0 puisque la précondition est satisfaite ou sinon la fonction n'est pas exécutée. On se trouve donc dans le cas d'un arrêt du calcul et le résultat obtenu est :

```
car = item(CHAÎNE_VIDE, 0) = CAR_VIDE
```

Par conséquent, la fonction rend VRAI lorsqu'elle est appelée avec car = CAR_VIDE sur une chaîne vide et FAUX sinon.

Finalement, en rassemblant ce que nous avons mis au point, on obtient l'algorithme ci-dessous :

*Algorithme 2 : **estDans** - Version 1.0*

```
Algorithme estDans
    # car est-il un caractère de ch entre les caractères de numéros
    # début et fin ?
Entrée
    ch         : CHAÎNE    # La cible de la recherche.
    début, fin : ENTIER    # L'intervalle de recherche.
    car        : CARACTÈRE # Le caractère cherché.
Résultat : BOOLÉEN
précondition
    car ≠ NUL
    ch ≠ NUL
    index_valide(ch, début)
    index_valide(ch, fin)
postcondition
```

© ENI Editions - All rigths reserved

```
    début > fin => Résultat = FAUX
    début = fin => Résultat = (car = item(ch, début))
    début < fin => Résultat =
        (
            (car = item(ch, début))
          ou sinon
            (estDans(ch, début+1, fin, car))
        )
fin estDans
```

La réalisation est claire :

```
réalisation
    si
        début > fin
    alors
        Résultat <- FAUX
    sinon si
        début = fin
    alors
        Résultat <- (car = item(ch, début))
    sinon
        Résultat <- estDans(ch, début+1, fin, car)
    fin si
```

On regarde si le caractère cherché est celui de numéro début. Sinon, on le cherche entre début + 1 et fin.

Remarquez les deux dernières clauses de la postcondition. La première donne le résultat calculé lorsque début = fin. La seconde indique le résultat calculé lorsque début < fin. Cependant, dans ce deuxième cas, on observe de nouveau le premier caractère, celui de numéro début. *Peut-on simplifier et écrire dans ce cas* :

```
# SOLUTION FAUSSE !
postcondition
    Résultat = estDans(ch, début+1, fin, car)
# FIN DE LA SOLUTION FAUSSE.
```

L'idée, ici, est que, comme début ≤ fin pour les deux dernières clauses de la postcondition, il suffit d'observer le premier caractère, celui de numéro début, puis ceux qui restent lorsque le premier test échoue.

Pouvez-vous expliquer pourquoi cette solution est fausse ?

Cette clause modifiée correspond au cas début < fin. Si le caractère cherché, c'est-à-dire `car`, est le premier caractère de la chaîne, la postcondition ne le « découvre » pas lorsque début < fin.

3. Exercices résolus

Exercice résolu 1 : Inverser une chaîne de caractères

Inverser une chaîne de caractères, c'est construire la chaîne de caractères égale à l'image miroir de la chaîne de départ. Autrement dit, on veut obtenir la chaîne de caractères qui est égale à la chaîne de départ quand on la lit de droite à gauche. Ainsi, par exemple, l'inversion de 'bille' donne 'ellib'.

Définir un algorithme qui inverse une chaîne de caractères.

Solution

On donne une chaîne ch. On veut construire la chaîne égale à son image miroir. Ainsi, par exemple, lorsque ch = 'Noël' on obtient **Résultat** = 'lëoN'.

On remarque d'abord que, lorsque la chaîne initiale ch est vide, ou lorsqu'elle est réduite à un seul caractère, elle est égale à son inverse. C'est la condition de terminaison de la récursion :

```
si longueur(ch) < 2 alors Résultat <- ch sinon ??? fin si
```

Le bloc de l'alternant qui commence à **sinon** indique comment inverser une chaîne qui comporte au moins deux caractères. *Comment faire ?* Nous disposons de deux stratégies équivalentes :

© ENI Editions - All rigths reserved

(s1) :

(i) : extraire le premier caractère : `N oël` ;

(ii) : inverser le reste de la chaîne : `lëo` ;

(iii) : recomposer une chaîne en plaçant le premier caractère à droite : `lëoN`.

(s2) :

(i) : extraire le dernier caractère : `Noë l` ;

(ii) : inverser le reste de la chaîne : `ëoN` ;

(iii) : recomposer une chaîne en plaçant le dernier caractère à gauche : `lëoN`.

Pour chacune de ces méthodes, la deuxième étape consiste à inverser une chaîne qui a un caractère de moins que ch. Il suffit donc d'utiliser les services du même algorithme et ainsi, à chaque appel de l'algorithme d'inversion, la longueur de la chaîne à traiter diminue strictement d'une unité. Comme la chaîne à traiter est de longueur finie, la longueur atteindra 1 et l'algorithme se terminera. À ce stade, nous avons une stratégie complète de résolution du problème posé :

```
inverser(ch : CHAÎNE) : CHAÎNE
    # L'image miroir de ch.
précondition
    ch ≠ NUL
réalisation
    si
        longueur < 2
    alors
        Résultat <- ch
    sinon
        Résultat <- (l'image miroir de ch privée de son premier
        caractère, augmentée à droite du premier caractère de ch)
    fin si
postcondition
    Résultat = (l'image miroir de ch)
fin inverser
```

Notons **fin** la fonction qui rend une chaîne égale à son argument, mais privée de son premier caractère. Ainsi, par exemple, **fin(**'Noël') = 'oël'.

Sa signature est :

```
Algorithme fin
    # la copie de ch sans son premier caractère.
Entrée
    ch  : CHAÎNE
Résultat : CHAÎNE
```

On accède à une copie du premier caractère d'une chaîne en utilisant la fonction **item** :

```
premier <- item(ch, index_min(ch))
```

Pour simplifier les notations, on peut appeler **premier** la fonction qui rend une copie du premier caractère de son argument. On a :

*Algorithme 3 : Définition de **premier***

```
Algorithme premier
    # Le premier caractère de ch.
Entrée
    ch : CHAÎNE
    Résultat : CHAÎNE
précondition
    ch ≠ NUL
postcondition
    Résultat = item(ch, index_min(ch))
fin premier
```

© ENI Editions - All rigths reserved

Cependant, nos stratégies de calcul prévoient, à l'étape (iii), la recomposition de deux chaînes de caractères, alors que **premier** rend un **CARACTÈRE**. Il faut donc convertir en une chaîne le premier caractère obtenu à l'étape (ii) avant de réaliser la recomposition. Soit **chaîne** la fonction qui prend un caractère en argument et qui rend la chaîne ne contenant que ce caractère. Cette fonction a déjà été définie au chapitre précédent.

Dans chaque stratégie, le résultat est recomposé, à l'étape (iii), en plaçant deux chaînes de caractères « bout-à-bout », l'une après l'autre. Une telle opération est appelée la *concaténation* des chaînes de caractères. Pour noter la concaténation de deux chaînes ch_1 et ch_2, on peut utiliser la notation fonctionnelle :

```
Résultat <- concaténer(ch_1, ch_2)
```

Cependant, les notations sont moins lourdes et plus lisibles en utilisant un opérateur infixe, disons ⊕, comme pour une addition :

```
Résultat <- ch_1 ⊕ ch_2
```

Ainsi, par exemple,

```
'Bon' ⊕ 'jour' = 'Bonjour'
```

La définition de l'algorithme de concaténation est la suivante :

Algorithme 4 : Définition de la concaténation des chaînes de caractères

```
Algorithme concaténer, infixe ⊕
    # La chaîne faite des caractères de ch_1 suivis de ceux de ch_2.
Entrée
    ch_1, ch_2 : CHAÎNE
Résultat : CHAÎNE
précondition
    ch_1 ≠ NUL
    ch_2 ≠ NUL
postcondition
    longueur(ch_2) = 0 => Résultat = ch_1
    longueur(ch_2) ≠ 0 => Résultat =
        (
        ch_1 à gauche de chaîne(premier(ch_2)) ⊕ fin(ch_2)
        )
fin concaténer
```

Autrement dit, le résultat est ch_1 lorsque ch_2 est vide. C'est ch_1 augmentée du premier caractère de ch_2, concaténée avec la fin de ch_2. Chaque invocation de l'algorithme a pour effet de placer un nouveau caractère à droite du résultat partiel obtenu à l'appel précédent. Ce nouveau caractère est une copie du premier caractère du second argument. Cette imbrication d'appels se termine puisque la longueur des chaînes en second argument diminue à chaque appel.

Il est aussi possible d'étendre le type **CARACTÈRE** en lui donnant l'opération de concaténation. Concaténer deux caractères, c'est former la chaîne faite des deux caractères concernés. Ainsi, par exemple, c_1 ⊕ c_2 = 'ab' lorsque c_1='a' et c_2='b'. C'est une opération *externe* en ce sens qu'elle rend un résultat, une chaîne de caractères, qui n'appartient pas au domaine de c_1 et c_2 qui sont des caractères. Cependant, cette nouvelle opération n'est pas strictement nécessaire puisqu'il est à présent toujours possible de transformer d'abord un caractère en chaîne, donc de transformer d'abord c_1 et c_2 en deux chaînes de caractères, puis de les concaténer en utilisant l'opération applicable aux chaînes :

```
variable
    c_1, c_2       : CARACTÈRE
    ch_1, ch_2, ch : CHAÎNE
...
    # Transformer c_1 et c_2 en chaînes.
    ch_1 <- chaîne(c_1) ; ch_2 <- chaîne(c_2)
    # Former la chaîne résultat de la concaténation.
    ch <- ch_1 ⊕ ch_2
...
```

Nous disposons à présent de tout ce dont nous avons besoin pour définir complètement la fonction **inverse**. C'est l'algorithme 5.

© ENI Editions - All rigths reserved

*Algorithme 5 : Définition de **inverse** - Version 1.0*

```
Algorithme inverse
    # L'image miroir de ch.
Entrée
    ch : CHAÎNE
Résultat : CHAÎNE
précondition
    ch ≠ NUL
postcondition
    longueur(ch) < 2 => Résultat = ch
    longueur(ch) ≥ 2 => Résultat =
        (
            inverse(fin(ch))
          ⊕
            chaîne(premier(ch))
    )
fin inverse
```

Étudions le « fonctionnement » de cet algorithme sur un exemple.

```
inverse('abc') = inverse('bc') ⊕ 'a'
inverse('abc') = (inverse('c') ⊕ 'b') ⊕ 'a'
inverse('abc') = ('c' ⊕ 'b') ⊕ 'a'
inverse('abc') = 'cb' ⊕ 'a'
inverse('abc') = 'cba'
```

La réalisation de inverse est simple :

```
réalisation
    si
        longueur(ch) < 2
    alors
        Résultat <- ch
    sinon
        Résultat <- inverse(fin(ch)) ⊕ chaîne(premier(ch))
    fin si
```

Exercice résolu 2 : Position d'un caractère dans une chaîne

On donne une chaîne de caractères ch et un caractère car quelconque.

Quelle est la position de car dans ch ?

Solution

La position du caractère est son numéro dans la chaîne. Ainsi, par exemple, avec les déclarations :

```
variable
    phrase : CHAÎNE[5] <- 'Il fait beau.'
    car_1  : CARACTÈRE <- 'f'
    car_2  : CARACTÈRE <- 'z'
```

on obtient les résultats :

```
position(phrase, car_1) = 8
position(phrase, car_2) = ???
```

La fonction rend 8 pour le caractère 'f' puisque les caractères de phrase sont numérotés à partir de 5. Il faut convenir d'un résultat lorsque le caractère dont on cherche la position n'est pas un constituant de la chaîne. Il est possible de commencer par une fonction qui n'est définie que pour des caractères dont on sait qu'ils composent la chaîne :

```
...
précondition
    ch ≠ NUL
```

© ENI Editions - All rigths reserved

```
    car ≠ NUL
    appartient(ch, car)
```

Le sous-algorithme **appartient** est un prédicat qui rend VRAI si et seulement si le caractère car est un constituant de la chaîne ch. Il a été défini au chapitre Structures élémentaires. Ainsi, le deuxième cas de l'exemple précédent ne peut plus se produire, mais il faut être certain que car compose la chaîne ch et donc s'en assurer indépendamment de **position**, avant d'utiliser ses services.

Soit alors **rang** le fonction qui donne la position d'un caractère dans une chaîne, mais recherchée à partir d'un caractère qui n'est pas nécessairement le premier. Nous avons déjà étudié une méthode similaire avec les fonctions **appartient** et **estDans**. La fonction **position** s'écrit :

*Algorithme 6 : Fonction **position** - Version 1.0*

```
Algorithme position
    # Le numéro de car dans ch.
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
Résultat : ENTIER
précondition
    ch  ≠ NUL
    car ≠ NUL
    appartient(ch, car)
réalisation
    Résultat <- rang(ch, index_min(ch), index_max(ch), car)
postcondition
    car = item(ch, Résultat)
fin position
```

La postcondition précise que le caractère recherché, car, est le caractère obtenu par copie de celui en position **Résultat** dans ch. Étudions à présent la fonction **rang**.

La signature et la précondition de cette fonction sont :

```
rang(ch : CHAÎNE ; début, fin : ENTIER ; car : CARACTÈRE) : ENTIER
    # Le numéro de car présent dans ch entre les caractères
    # de numéros début et fin.
précondition
    ch ≠ NUL
    car ≠ NUL
    estDans(ch, début, fin, car)
    index_valide(ch, début)
    index_valide(ch, fin)
    début ≤ fin
```

Avec ces hypothèses, le rang de car est début lorsqu'il est le premier caractère de la partie de la chaîne observée :

```
postcondition
    car = item(ch, début) => Résultat = début
    ...
```

Lorsqu'il n'est pas le premier caractère, il est donc présent dans la chaîne constituée des caractères de numéros compris entre début + 1 et fin :

```
    car ≠ item(ch, début) => Résultat = rang(ch, début+1, fin, car)
```

Le nombre d'états visités lors de cette recherche est majoré par fin – début + 1, qui est le nombre de caractères de la partie de la chaîne à observer. Chaque appel de rang réduit cette « *distance à la solution* » d'une unité. Par conséquent, l'algorithme termine avec un résultat tel que début ≤ **Résultat** ≤ fin. Les spécifications complètes de l'algorithme **rang** sont :

*Algorithme 7 : Spécifications de **rang** - Version 1.0*

```
Algorithme rang
```


© ENI Editions - All rigths reserved

```
    # Le numéro de car présent dans ch entre les caractères
    # de numéros début et fin.
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
    début, fin : ENTIER
Résultat : ENTIER
précondition
    ch ≠ NUL
    car ≠ NUL
    estDans(ch, début, fin, car)
    index_valide(ch, début)
    index_valide(ch, fin)
    début ≤ fin
postcondition
    car = item(ch, début) => Résultat = début
    car ≠ item(ch, début) => Résultat = rang(ch, début+1, fin, car)
fin rang
```

Il est possible de relâcher la contrainte imposée par la troisième clause de la précondition. Dans ce cas, on veut déterminer le rang d'un caractère dont on ne sait pas s'il compose la partie de la chaîne cible de la recherche. S'il compose cette partie de la chaîne, nous sommes ramenés au cas précédent ; sinon, il faut convenir d'un résultat qui signera l'absence du caractère. Un rang invalide pour un caractère d'une chaîne ch est un rang strictement inférieur ou strictement supérieur aux index minimum et maximum respectivement. Convenons que la fonction renvoie la valeur de la constante égale à **index_min**(ch) - 1 lorsque le caractère n'est pas trouvé.

Lorsque le caractère cherché est le premier, le raisonnement est le même que précédemment ; sinon, il faut séparer les cas et déterminer s'il est présent ou non dans la chaîne. S'il appartient à la chaîne sans être le premier, on est revenu au cas précédent. S'il n'appartient pas à la chaîne, la fonction rend la valeur convenue. Il est donc possible de donner les spécifications de la deuxième version de **rang** :

Algorithme 8 : Spécifications de rang - Version 2.0

```
Algorithme rang
    # Le numéro de car s'il est présent dans ch entre les caractères
    # de numéros début et fin. ABSENT s'il n'y est pas.
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
    début, fin : ENTIER
Résultat : ENTIER
constante
    ABSENT : ENTIER <- index_min(ch) - 1
précondition
    ch ≠ NUL
    car ≠ NUL
    index_valide(ch, début)
    index_valide(ch, fin)
postcondition
    non estDans(ch, début, fin, car) => Résultat = ABSENT
        estDans(ch, début, fin, car) et alors
            (
                item(ch, début) = car => Résultat = début
              ou sinon
                item(ch, début) ≠ car => Résultat = rang(ch, début+1,
                fin, car)
            )
fin rang
```

Il n'est pas nécessaire d'utiliser le prédicat **estDans** pour spécifier cet algorithme. La postcondition devient alors :

```
...
postcondition
    début = fin =>
        (
            item(ch, début) = car => Résultat = début
          ou sinon
```

© ENI Editions - All rigths reserved

```
            item(ch, début) ≠ car => Résultat = ABSENT
        )
    début < fin =>
        (
            item(ch, début) = car => Résultat = début
          ou sinon
            Résultat = rang(ch, début+1, fin, car)
        )
...
```

Cette fonction permet donc de déterminer la position d'une occurrence quelconque d'un caractère dans une chaîne. C'est souvent utile, comme dans l'exercice suivant :

Exercice 3 : Nombre de mots

On donne une phrase. Compter les mots de cette phrase. Pour simplifier le problème, on suppose qu'une espace unique sépare deux mots et que c'est le seul séparateur qui distingue les mots.

Écrire un algorithme qui compte les mots d'une phrase.

Les hypothèses proposées simplifient beaucoup le problème. Ainsi, par exemple, on suppose que l'on ne rencontre pas de phrase comme 'Voici deux exemples : le premier ...' dans laquelle les trois caractères ' : ' séparent deux mots. Simplifions encore le problème, en supposant que la phrase étudiée est « bien formée », c'est-à-dire qu'elle ne comporte pas d'espace en début et en fin et qu'elle n'a pas d'espace redoublée. Avec ces hypothèses, compter le nombre de mots, c'est compter le nombre d'espaces :

```
nombreDeMots <- nombreEspaces + 1
```

Le problème simplifié se ramène donc à compter les espaces d'une phrase bien formée.

```
# Compter les mots d'une phrase bien formée.
nombreDeMots <- nombreEspaces
    (
        phrase,            # La cible du comptage
        index_min(phrase), # Numéro du premier caractère observé.
        index_max(phrase)  # Numéro du dernier caractère observé.
    ) + 1
```

Cependant, compter les espaces est le même problème que compter un caractère quelconque. On obtient donc une solution plus générale en remplaçant la fonction **nombreEspaces** par une fonction **nombreDeCaractères** qui compte les occurrences d'un caractère quelconque.

```
# Compter les mots d'une phrase bien formée.
constante
    ESPACE : CARACTÈRE <- ' '
nombreDeMots <- nombreDeCaractères
    (
        phrase,            # La cible du comptage
        index_min(phrase), # Numéro du premier caractère observé.
        index_max(phrase), # Numéro du dernier caractère observé.
        ESPACE             # le caractère à compter.
    ) + 1
```

Le dernier paramètre est le caractère dont on compte les occurrences dans la phrase. Oublions alors le problème initial. On cherche une solution au problème plus général suivant :

On donne une chaîne de caractères ch et un caractère car. Faire un algorithme qui détermine le nombre d'occurrences de car dans ch.

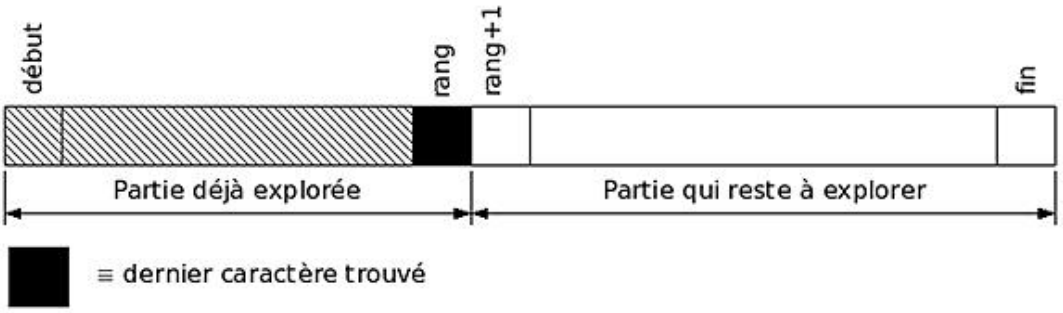
Nous savons déjà trouver la première occurrence d'un caractère donné dans une chaîne, à l'aide de la fonction **rang**. Lorsque la fonction rend un numéro de caractère invalide, c'est que le caractère cherché n'est pas présent dans la chaîne. On obtient donc :

```
...
si
    rang(ch, début, fin, car) = ABSENT
alors
    # Pas de car dans ch entre début et fin.
    Résultat <- 0
```


© ENI Editions - All rigths reserved

```
sinon
    ???
fin si
```

Dans le cas contraire, on vient de trouver une occurrence de car en position **rang**. Le nombre total d'occurrences de car est donc celle que l'on vient de trouver, augmentée du nombre d'occurrences entre rang + 1 et fin. La figure ci-dessous illustre cette situation.



On vient de découvrir un exemplaire du caractère car cherché en position rang. La partie qui reste à observer s'étend depuis le caractère de numéro rang + 1 jusqu'au caractère de numéro fin. Cette partie contient **nombreDeCaractères** (ch, rang + 1, fin, car) caractères cherchés. On obtient donc :

```
...
sinon
    Résultat <- 1 + nombreDeCaractères(ch, rang+1, fin, car)
fin si
...
```

D'où la définition de la fonction **nombreDeCaractères** :

Algorithme 9 : Compter les occurrences d'un caractère dans une chaîne - Version 1.0

```
Algorithme nombreDeCaractères
    # Le nombre d'occurrences de car dans ch entre les caractères de
    # numéros début et fin.
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
    début, fin : ENTIER
Résultat : ENTIER
précondition
    ch  ≠ NUL
    car ≠ NUL
    index_valide(ch, début)
    index_valide(ch, fin)
constante
    ABSENT : ENTIER <- index_min(ch) - 1
réalisation
    si
        rang(ch, début, fin, car) = ABSENT
    alors
        # Pas de car dans ch entre début et fin.
        Résultat <- 0
    sinon
        Résultat <- 1 + nombreDeCaractères(ch, rang+1, fin, car)
    fin si
postcondition
    rang(ch, début, fin, car) = ABSENT => Résultat = 0
    rang(ch, début, fin, car) ≠ ABSENT => Résultat = 1 +
         nombreDeCaractères(ch, rang(ch, début, fin, car)
         + 1, fin, car)
fin nombreDeCaractères
```

© ENI Editions - All rigths reserved

4. Exercices

Exercice 4 : Transformation des minuscules en majuscules

On donne un texte dont certains caractères sont des majuscules, d'autres sont des minuscules, d'autres enfin des caractères qui ne sont pas alphabétiques, comme des chiffres, des signes de ponctuation...

1. Écrire un algorithme qui transforme tous les caractères minuscules en majuscules.

On pourra supposer que l'exercice ne concerne que le parcours du texte. Ainsi, on accepte que les modules logiciels nécessaires à la reconnaissance des types de caractères et à la transformation d'une minuscule en majuscule sont connus et font partie du répertoire d'instructions. Ils ont d'ailleurs déjà été proposés en exercices au chapitre précédent. Peut-être est-il temps d'étudier les solutions de ces exercices si cela n'a pas encore été fait, ou sinon :

2. Donner au moins les spécifications des modules utilisés.

Dans l'exercice suivant, on change les conventions adoptées précédemment. Il faut donc bien veiller à ne pas utiliser, pour le résoudre, des hypothèses propres au contexte précédent.

Un caractère est, à présent, un nombre entier positif ou nul. Dans la suite, un caractère est donc confondu avec son code numérique. Le domaine des valeurs d'un tel entier n'est pas précisé davantage. On peut y penser en supposant, par exemple, que le domaine dépend de la machine utilisée. Le type de données **CARACTÈRE** permet de définir une variable de type caractère.

L'un de ces caractères est un code particulier. Il est désigné par FIN_CH. Ce caractère *n'est jamais un caractère significatif* dans une chaîne. Il n'est utilisé que pour marquer la terminaison d'une chaîne. Ainsi, toute donnée de type **CHAÎNE** contient, comme dernier caractère, celui de code FIN_CH qui en marque la fin. Les caractères d'une chaîne sont numérotés en séquence à partir de 1 pour le premier et nous ne disposons pas des facilités de numérotation des sections précédentes.

L'accesseur **item** défini précédemment reste disponible dans le répertoire d'instructions. C'est une fonction qui prend en entrée une chaîne de caractères bien formée, c'est-à-dire terminée par FIN_CH, et un entier naturel rang. Elle rend le caractère de la chaîne qui occupe le rang donné, comme dans les sections précédentes.

Exercice 5 : Longueur d'une chaîne de caractères

On s'intéresse au calcul du nombre de caractères d'une chaîne de caractères.

*1. Définir la signature et donner les spécifications de la fonction **longueur** qui prend en entrée une chaîne de caractères bien formée et qui retourne le nombre de caractères de cette chaîne.*

2. Écrire l'algorithme récursif de cette fonction.

Le caractère FIN_CH n'est pas compté comme un caractère de la chaîne.

Remarquez qu'il n'a pas été dit qu'une chaîne de caractères est un tableau d'entiers, comme en langage C par exemple. En fait, on ne dit rien et vous ne devez faire aucune hypothèse supplémentaire sur l'implémentation d'une chaîne de caractères.

Exercice 6 : Normalisation d'une chaîne de caractères

Normaliser une chaîne de caractères, c'est la débarrasser de ses espaces inutiles. Une espace est inutile lorsqu'elle est le premier ou le dernier caractère de la chaîne, ou qu'elle est redoublée dans le corps de la chaîne. Ainsi, par exemple, la chaîne :

' Il fait beau aujourd'hui. '

est normalisée en 'Il fait beau aujourd'hui.'.

Faire un algorithme qui normalise une chaîne de caractères.

On appelle *palindrome* un texte qui est le même que son image miroir. Ainsi, par exemple, les mots 'LAVAL', 'non', '26762' sont des palindromes. Autrement dit, un palindrome se lit de droite à gauche comme de gauche à droite.

Avec cette définition, 'Laval' ou 'non !' ne sont plus des palindromes. Pour le premier, la majuscule initiale et pour le second l'espace et le point d'exclamation viennent perturber l'image miroir du mot. De même, la phrase 'Élu par cette crapule' serait un palindrome si la majuscule accentuée au début de la phrase était la lettre 'e' et si les espaces étaient supprimées. Existe-t-il des phrases palindromes ? Le problème est compliqué par les caractères séparateurs, comme l'espace, les signes de ponctuation... De plus, une phrase bien formée se termine toujours par un point, mais son premier caractère n'est jamais un point, ce qui répond à la question.

Généralisons donc cette définition en l'étendant pour simplifier le problème. Un palindrome est recherché parmi les chaînes de caractères alphanumériques dont les caractères alphabétiques sont en majuscules, ou en minuscules, comme on voudra, et dans lesquelles les caractères accentués ont été remplacés par leurs équivalents sans accent.

© ENI Editions - All rigths reserved

’Élu par cette crapule’ devient ’ELUPARCETTECRAPULE’ ou ’eluparcettecrapule’ et c’est un palindrome.

Reconnaître un palindrome consiste donc à réaliser quatre traitements sur le texte à analyser :

- filtrer le texte afin de ne conserver que des caractères alphanumériques ;
- remplacer les caractères accentués par leur équivalent sans accent ;
- remplacer chaque caractère alphabétique par sa majuscule ou sa minuscule ;
- vérifier que le texte filtré est égal à son image miroir.

Voici quelques exemples de « phrases » qui sont des palindromes après filtrage.

Exemples

- Ésope reste ici et se repose ;
- Élu par cette crapule ;
- Laval ;
- 1754571 ;
- 10000000000000000001 ;
- Non.

Exercice 7 : Reconnaître un palindrome

Faire un algorithme qui reconnaît un palindrome.

© ENI Editions - All rigths reserved

Les nombres et la récursivité

Les nombres fournissent de nombreux exemples simples de récursivité. En fait, les définitions mathématiques sont souvent récursives. Cette section vous propose quelques exercices sur ce sujet. Ceux qui ne peuvent se réconcilier avec les Mathématiques peuvent passer à la section suivante.

1. Arithmétique

Soit $\mathbb{Z}$ l'ensemble des entiers : $\mathbb{Z}$ = { ..., -2, -1, 0, 1, 2, ...}. On suppose que les seules opérations actuellement définies sur $\mathbb{Z}$ sont **successeur**, qui rend le successeur d'un entier dans la liste ordonnée de ses éléments et **prédécesseur** la fonction duale de la précédente.

Exercice 8 : Arithmétique sur les entiers

1. Écrire l'algorithme de l'addition de deux entiers.

2. Écrire l'algorithme qui rend l'opposé d'un entier.

La différence de deux entiers est la somme du premier avec l'opposé du deuxième.

3. Écrire l'algorithme qui calcule la différence de deux entiers.

Le produit de deux entiers se calcule en effectuant une addition répétée. Ainsi, par exemple, 7 x 5 est la somme de 5 termes égaux à 7.

4. Écrire les algorithmes de la multiplication et de la division.

L'exercice suivant est plus difficile et peut être ignoré en phase d'apprentissage.

La fonction de ACKERMANN généralisée est une fonction permettant de calculer, suivant la valeur de l'un de ses paramètres, la somme, le produit ou une puissance de deux entiers. Elle est définie ainsi :

```
A(0, a, b) = a+1 ;
A(1, a, 0) = a ;
A(2, a, 0) = 0 ;
A(3, a, 0) = 1 ;
A(n, a, 0) = 2 pour tout n>3 ;
A(n, a, b) = A(n-1, A(n, a, b-1), a) sinon.
```

Il est alors possible de vérifier que :

```
A(1, a, b) = a+b ;
A(2, a, b) = axb ;
A(3, a, b) = a^b ;
A(4, a, b)= 2^(a^b) .
```

*Exercice 9 : Fonction de **ACKERMANN***

*Écrire l'algorithme qui calcule la valeur rendue par la fonction de **ACKERMANN**.*

2. Factorielle et autres exercices usés

Si vous avez cru y échapper, c'est raté avec cette section.

Exercice 10 : Factorielle d'un entier positif

Soit n un entier naturel, c'est-à-dire positif ou nul. On appelle factorielle de l'entier n, que l'on note n!, l'entier défini par :

```
0! = 1! = 1 ;
n! = n x(n-1)x(n-2)x...x2 pour tout n > 1
```

*Écrire l'algorithme de la fonction **factorielle** qui rend n! pour tout n positif.*

© ENI Editions - All rigths reserved

Exercice 11 : Suites de LUCAS

La suite de FIBONACCI est définie par :

$u_0 = u_1 = 1$;
$u_n = u_{n-1} + u_{n-2}$ pour tout $n > 1$

1. Écrire l'algorithme de calcul du terme d'indice n ≥ 0 de cette suite.

Les deux premiers termes de la suite sont égaux à 1. En donnant deux valeurs consécutives aux deux premiers termes, on obtient une suite de LUCAS.

2. Écrire l'algorithme de la fonction qui calcule le terme de rang n d'une suite de LUCAS.

3. Fractions

Le type **FRACTION** a déjà été défini au chapitre Structures élémentaires de la façon suivante :

```
type
    FRACTION
structure
    numérateur   : ENTIER
    dénominateur : ENTIER
invariant
    dénominateur ≠ 0
fin FRACTION
```

Exercice 12 : Calculs sur les fractions

On dit qu'une fraction est normalisée lorsque son dénominateur est strictement positif. Ainsi, par exemple, la fraction (5 ; -7) n'est pas normalisée, mais la fraction normalisée équivalente est (-5 ; 7).

1. Écrire une fonction qui prend en entrée une fraction et qui rend la fraction normalisée équivalente.

2. Écrire l'algorithme de la fonction qui prend en entrée une fraction et qui rend la fraction normalisée irréductible équivalente.

*3. Écrire la fonction **addition** qui prend en entrée deux fractions et qui rend la fraction irréductible normalisée équivalente à leur somme.*

La différence de deux nombres est la somme du premier avec l'opposé du second.

*4. Écrire la fonction **soustraction** qui prend en entrée deux fractions et qui retourne la fraction irréductible normalisée équivalente à leur différence.*

*5. Écrire de même la fonction **multiplication** de deux fractions.*

Le quotient de deux nombres est le produit du premier par l'inverse du second.

*6. Écrire la fonction **division** qui prend en entrée deux fractions et qui retourne la fraction irréductible normalisée équivalente à leur quotient.*

La récursivité n'intervient pas d'une façon apparente dans cet exercice. En fait, elle s'impose avec ce que nous savons, pour réduire une fraction. En effet, il est nécessaire de calculer le Plus Grand Commun Diviseur (pgcd) de deux entiers pour simplifier une fraction. Le pgcd de deux entiers s'obtient, par exemple, par l'algorithme de EUCLIDE. Voici un exemple de calcul du pgcd de deux entiers d'où vous aurez à déduire la méthode de résolution de la simplification de deux fractions.

Exemple

Soit à calculer le pgcd de 125 et 15. On divise d'abord 125 par 15. Le quotient entier est 8 et le reste 5. On divise alors le diviseur 15 par le reste 5. Le nouveau quotient est 3 et le reste est nul. On en déduit que le pgcd de 125 et de 15 est 5. C'est le dernier reste non nul dans la suite des divisions où, dans chacune, le diviseur de la division précédente devient le dividende et le reste de la division précédente devient le diviseur.

4. Fonction réelle

© ENI Editions - All rigths reserved

Exercice 13 : Puissance d'un nombre réel

Écrire l'algorithme d'une fonction qui rend x^n pour tout x réel et n entier.

0^0 n'a pas de sens.

© ENI Editions - All rigths reserved

Nombres et chaînes de caractères : édition d'un entier

On rencontre souvent le problème de l'*édition d'un entier* dans différents contextes. Nous aurons l'occasion de le retrouver dans les chapitres ultérieurs. Cette section commence l'étude de ce problème.

Il ne faut pas confondre la *valeur* d'un nombre et sa *représentation*. Il existe différentes représentations d'une même valeur. En voici deux :

- cent vingt-cinq ;
- 125

Dans la première forme, la valeur dont il s'agit est représentée par une chaîne de caractères qui énonce des mots de la langue française. On dira plus simplement que la valeur est représentée en lettres. La seconde forme représente la même valeur par une chaîne de trois caractères qui sont des chiffres de la numération en base dix. On dira que la valeur est représentée en chiffres. Il est aussi possible d'utiliser d'autres suites de caractères pour représenter la même valeur :

- 175_{huit} ;
- $7D_{seize}$;
- 1111101_{deux}.

La première forme représente la valeur du même nombre exprimée par des chiffres de la numération en base huit. La deuxième forme la représente en base seize et la dernière forme en base deux.

Définition

Éditer un entier c'est construire une chaîne de caractères qui représente la valeur de l'entier dans une base de numération B donnée.

Ainsi, éditer la valeur précédente en base B = dix, c'est construire la chaîne de trois caractères '125'. L'éditer en base B = seize, c'est construire la chaîne de caractères '7D'.

Exercice 14 : Édition d'un entier

Écrire l'algorithme qui édite un entier positif en base B.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a introduit la notion de traitements répétés. On élabore une définition des changements des états du logiciel au lieu de dire explicitement les transformations à faire subir aux données. On peut ainsi définir les algorithmes en énonçant des clauses sans effet de bord pour préciser les précondition, postcondition et invariant de leur spécification. Les chapitres suivants utiliseront intensivement la récursivité. C'est certainement une notion difficile à aborder pour une initiation. Il est tout à fait possible de s'en passer et d'exprimer les spécifications en utilisant des formules empruntées aux Mathématiques ou à la langue française. Il faut alors veiller à rester rigoureux.

© ENI Editions - All rigths reserved

Introduction

Un traitement *itératif* est un traitement *fait ou répété plusieurs fois* selon le « PETIT LAROUSSE ». Ce terme est synonyme de *fréquentatif* qui *se dit d'un verbe qui marque une action fréquemment répétée*, comme clignoter (...). Nous avons déjà vu comment construire des algorithmes qui définissent un traitement répété au chapitre Récursivité, qui a étudié la récursivité. Ici, nous retrouvons l'instruction d'affectation pour construire des algorithmes de transformation au cours desquels le système passe de l'état initial à l'état final en traversant une succession finie d'états caractérisés par une propriété commune. Nous montrons comment construire ce type d'algorithmes d'une façon raisonnée et systématique.

La section Premiers exemples de construction d'itérations introduit la méthode par quelques exemples simples. La section Itérer dans un tableau étudie différentes situations dans lesquelles l'itération est utilisée sur les composantes d'un tableau. La section Algorithme ou programme ? propose des exercices d'application au cours desquels il devient possible de donner des solutions itératives à des exercices déjà résolus récursivement. Le chapitre se conclut par quelques notes bibliographiques et un résumé.

© ENI Editions - All rigths reserved

Premiers exemples de construction d'itérations

Ces exemples sont simples. Le premier, la construction d'une table de multiplication, est même trivial. C'est qu'il s'agit d'un premier exposé pour introduire les raisonnements permettant de construire une itération : il faut éviter de multiplier les difficultés. C'est pourquoi on commence par se concentrer sur les étapes de la construction et du raisonnement sur un problème élémentaire, déjà abordé naïvement au chapitre précédent. Le second exemple explore un tableau dont les données sont des instances d'un type structuré. La section se termine ensuite par quelques exercices d'application de même nature.

1. La table de multiplication

a. Le problème

On veut préparer un algorithme qui remplit un tableau avec les résultats de la table de multiplication par un entier n positif ou nul. Le résultat à obtenir est semblable à celui présenté pour la table de multiplication par 9, au chapitre précédent. Il s'agit, cette fois, de préparer une solution qui présente les étapes du calcul des éléments du tableau. Ce tableau est encore limité à 11 cases, numérotées de 0 à 10, mais il pourrait comporter bien plus de cases, sans que l'algorithme proposé ne s'en trouve modifié. Voyons cela.

Le tableau des résultats à obtenir est :

```
...
constante
    MIN : ENTIER <- 0  # Numéro de la première case du tableau.
    MAX : ENTIER <- 10 # Numéro de la dernière case.
variable
    table : TABLEAU[ENTIER][MIN, MAX]
...
```

Ces instructions déclarent un tableau appelé table qui contiendra MAX - MIN + 1 entiers, entre les cases de numéros MIN et MAX. Pour remplir table avec la table de multiplication par 4, par exemple, nous écrirons :

```
...
    tableDe(table, 4)
...
```

Il reste à définir l'algorithme de la procédure **tableDe** qui initialise cette table. C'est une procédure qui prend en entrée un entier naturel n et qui initialise un tableau contenant les résultats de la table de multiplication par n. Elle est définie par :

Algorithme 1 : Table de multiplication - Version 1.0

```
Algorithme tableDe
    # La table de multiplication par n.
Entrée
    n : ENTIER          # L'entier dont on veut la table.
    t : TABLEAU[ENTIER] # La table à initialiser.
précondition
    n ≥ 0
postcondition
    ancien(n) = n       # n n'est pas modifié.
    (∀i, 0 ≤ i ≤ 10)(t[i] = i x n)
fin tableDe
```

L'expression de la postcondition définit précisément le comportement de la procédure. La première clause indique que la procédure ne modifie pas le paramètre n. La seconde clause exprime l'effet de la procédure sur la table par une expression mathématique. Dans ce cas, i est une variable mathématique muette. Il est clair qu'elle n'est pas une variable au sens algorithmique du terme. Elle n'est que le marque place d'une valeur qui modifie un état. La clause exprime que, pour toute valeur de i comprise entre les numéros extrêmes des cases du tableau, la case de numéro i contient la valeur i x n. Ainsi, la case de numéro 0 contient 0 x n = 0, la case suivante contient 1 x n = n et ainsi de suite jusqu'à la case de numéro 10. Une autre solution exprime la postcondition à l'aide d'un nouveau prédicat, **estTableDe**, qui rend VRAI si et seulement si le tableau qu'il reçoit en paramètre contient, entre deux de ses cases désignées, les résultats de la table de multiplication par un entier désigné :

© ENI Editions - All rigths reserved

```
postcondition
    ancien(n) = n       # n n'est pas modifié.
    estTableDe(t, index_min(t), index_max(t), n)
```

On exprime ainsi que le tableau t contient, entre ses cases de numéros rendus par **index_min** et **index_max**, les résultats de la table de multiplication par n. Bien entendu, ce nouveau prédicat doit être défini. C'est fait par l'algorithme ci-dessous.

*Algorithme 2 : Spécifications du prédicat **estTableDe***

```
Algorithme estTableDe
    # t[début .. fin] contient-il la table de n ?
Entrée
    t : TABLEAU[ENTIER]
    début, fin, n : ENTIER
Résultat : BOOLÉEN
précondition
    t ≠ NUL
    index_valide(t, début)
    index_valide(t, fin)
    n ≥ 0
postcondition
début > fin => Résultat = FAUX
début = fin => Résultat = (t[début] = (début - index_min(t)) x n)
début < fin => Résultat =
    (
        (t[début] = (début - index_min(t)) x n)
      et alors
        (estTableDe(t, début+1, fin, n))
    )
fin estTableDe
```

L'algorithme définit cette fois une fonction dont le résultat est **BOOLÉEN**. La postcondition n'utilise plus le symbolisme mathématique, mais une définition récursive du résultat. Lorsque les bornes `début` et `fin` du calcul sont égales, le résultat est la valeur booléenne de l'expression (début - **index_min**(t)) x n. C'est 0 pour début = **index_min**(t), n pour la valeur suivante et ainsi de suite. Lorsque les bornes ne sont pas égales, c'est la même valeur dans la première case et les cases suivantes contiennent le reste de la table de multiplication par n.

Cet algorithme est remarquable par sa généralité. Il exprime que les cases de numéros `début`, `début+1`, ..., `fin` contiennent respectivement les valeurs (début-**index_min**(t))xn, (début-**index_min**(t)+1)xn... Autrement dit, ces cases contiennent bien l'extrait de la table de multiplication par n qui correspond à la table complète lorsque début=**index_min**(t) et fin=**index_max**(t).

La preuve de correction n'est pas difficile à établir par un raisonnement identique à ceux pratiqués au chapitre précédent. En particulier, il n'est pas difficile de calculer la distance à la solution, autrement dit le nombre d'appels récursifs avant la terminaison de l'algorithme.

Il n'est pas certain que l'expression de la postcondition de l'algorithme 1 soit plus compliquée ou absconse que celle qui utilise l'algorithme 2. Pour ce qui me concerne, je préfère et de loin la première forme de spécification. Elle est compréhensible d'emblée, sans effort avec une culture mathématique de niveau élémentaire. La deuxième forme me semble bien plus difficile et incomparablement moins évidente. Pourtant, il FAUT spécifier précisément et clairement nos algorithmes : nous ne pouvons y échapper. Il faut donc choisir : les Mathématiques ou... l'algorithmique. La suite de ce chapitre va le montrer.

Il reste à réaliser l'algorithme **tableDe**.

Si nous savons remplir la case de numéro i - 1 du tableau t, nous savons aussi remplir la case de numéro i en remplaçant i - 1 par i. Soit (1) cette assertion :

« On sait initialiser la case numéro i quand on sait initialiser la case numéro i - 1. »

Peu importe pour l'instant la valeur d'initialisation.

Or, nous savons initialiser la première case, celle de numéro MIN : sa valeur est v_0=0. Voyons pourquoi ceci résout le problème.

- connaissant la valeur v_0 à placer dans le tableau dans la case de numéro i - 1 = MIN, on connaît la valeur v_1 à placer dans la case de numéro i = MIN + 1 d'après l'assertion (1) ;

© ENI Editions - All rigths reserved

- connaissant la valeur v_1 à placer dans le tableau dans la case de numéro i - 1 = MIN + 1, on connaît la valeur v_2 à placer dans la case de numéro i = MIN + 2 d'après l'assertion (1) ;

- connaissant la valeur v_2 à placer dans le tableau dans la case de numéro i - 1 = MIN + 2, on connaît la valeur v_3 à placer dans la case de numéro i = MIN + 3 d'après l'assertion (1).

Cette suite est une itération. Les mêmes instructions sont renouvelées, répétées plusieurs fois, sur des jeux de données différents mais liés. La valeur $v_0 = 0$ est connue. Les valeurs v_1, v_2... s'en déduisent. À chaque étape, la valeur de i avance pour que la position qu'il repérait avant son changement de valeur recule. Toute ceci est obtenu à l'aide de l'assertion (1) qui exprime que la connaissance des traitements à l'étape numéro i se déduit de la connaissance des traitements à l'étape i - 1. Ainsi, comme nous savons résoudre le problème à l'étape i = 0, nous savons le résoudre à l'étape suivante i = 1. Il reste à comprendre comment calculer la valeur v_i quand on connaît la valeur précédente, v_{i-1}.

Dans la case de numéro i - 1 ≥ 0 du tableau t, on trouve la valeur $v_{i-1} = (i - 1) \times n$ par hypothèse. La case de numéro i doit contenir $v_i = i \times n$. On a :

$$v_{i-1} = (i-1) \times n$$
$$v_{i-1} = i \times n - n$$
$$v_{i-1} = v_i - n$$

D'où :

$$v_i = v_{i-1} + n$$

Par conséquent, v_i, i > 0, est la valeur précédente augmentée de n : « *on compte de n en n* ». Ainsi, on obtient :

$$v_0 = 0$$
$$v_i = v_{i-1} + n \text{ lorsque } 0 < i \leq 10$$

Pour passer de la valeur d'une case à la suivante, il suffit d'ajouter n. Ainsi, à chaque étape on a, en désignant par `valeur` la valeur de la nouvelle case à remplir :

```
valeur <- valeur + n
```

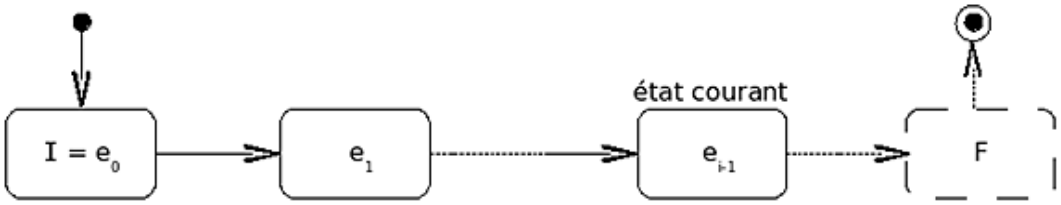
Nous venons de redécouvrir que réciter la table de multiplication par n, c'est compter de n en n. Tout ceci est assez évident pour qui connaît ses tables de multiplication, mais… patience.

b. Construction de l'itération

L'itération se construit en cinq étapes ordonnées, toujours les mêmes.

Faire une hypothèse sur l'état actuel

Les chapitres précédents ont déjà insisté sur le fait qu'un algorithme décrit une suite de transformations à faire subir aux données pour obtenir un résultat attendu. Ces transformations font évoluer l'état du système représenté par les valeurs des variables. Cependant, les notations des transformations décrites par l'algorithme ne disent pas l'état qui en résulte. Il est donc nécessaire d'apporter, dans la description, les éléments indispensables à la caractérisation complète des états parcourus par le système depuis l'état initial jusqu'à l'état final dans lequel le résultat est obtenu. Dans la première étape de construction de l'itération, on fait une hypothèse sur un état intermédiaire du système. On suppose que le travail a déjà été partiellement réalisé. Le système a évolué, depuis l'état initial I jusqu'à un état intermédiaire e_{i-1} qu'il s'agit de caractériser. La figure ci-dessous illustre ce point.



Pour caractériser l'état intermédiaire e_{i-1} déjà atteint, on se réfère à ce qui a été découvert au paragraphe précédent et à l'assertion (1) qui a permis de construire l'analyse.

© ENI Editions - All rigths reserved

Hypothèse (H) : les cases de t jusqu'à la case de numéro i - 1, comprise, ont déjà été initialisées. La valeur dans la case de numéro i - 1 est (i - 1) x n. La variable `valeur` contient la prochaine valeur i x n à ranger.

Voir si c'est fini

Dans quelles circonstances la situation décrite par cette hypothèse résout-elle entièrement le problème posé ? C'est fini lorsque l'état intermédiaire e_{i-1} est l'état final F. On a 11 cases numérotées de 0 à 10 à initialiser et les cases de numéros 0 à i - 1 ont déjà été initialisées d'après l'hypothèse **(H)**. C'est fini s'il n'existe plus de case à initialiser, donc lorsque i - 1 = 10. Il est donc possible d'écrire :

```
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
si
    i - 1 = 10
alors
    c'est fini
sinon
...
```

Se rapprocher de l'état final

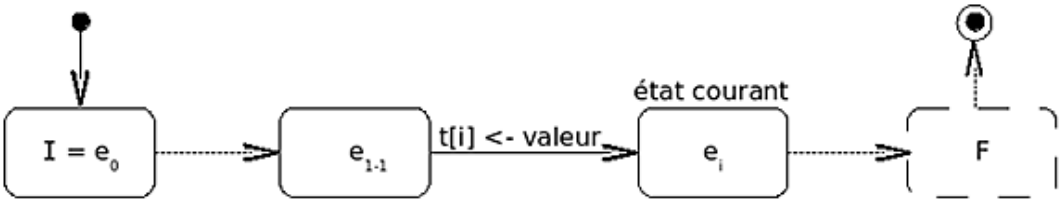
Sinon, il faut chercher un nouvel état intermédiaire plus « proche » de l'état final F. Cependant, nous ne pouvons pas le faire sans précaution. Il nous faut trouver un nouvel état intermédiaire qui vérifie l'hypothèse **(H)**, mais avec une plus grande valeur de `i`. Nous avons la solution pour `i` éléments, numérotés de 0 à `i - 1` et nous la cherchons pour `i + 1` éléments. Ceci revient à initialiser une nouvelle case, celle de numéro `i`, dans le tableau t. Actuellement, la variable `valeur` contient la prochaine valeur à déposer dans t, dans la case numéro `i`. On a donc :

```
sinon
    t[i] <- valeur
```

Ainsi, les valeurs des variables qui représentent l'état du système viennent d'être modifiées. Par conséquent, l'état du système logiciel vient de changer. Il est passé de l'état e_{i-1} bien caractérisé par l'analyse du paragraphe précédent, à un état e_i dont on ne sait encore rien. Décrivons le :

```
# On a initialisé les cases de t jusqu'à la case de numéro i
# comprise. valeur = i x n
```

Ce n'est pas tout à fait l'état décrit par l'hypothèse **(H)**. Elle fait référence à la case de numéro `i - 1`. Nous en sommes à la case `i`. Elle décrit la situation représentée par la figure précédente. Nous sommes dans une nouvelle situation, représentée par la figure suivante :



Pour retrouver la situation décrite par l'hypothèse **(H)**, il suffit de changer la valeur de `i`. Si la valeur de `i` avance, la position qu'il repérait recule :

```
...
sinon
    t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
```

Ce n'est pas encore **(H)** puisque, dans cet état intermédiaire, `valeur = (i - 1) x n` alors que **(H)** prévoit que `valeur = i x n`. Reprenons donc la description de l'état et corrigeons cette situation :

© ENI Editions - All rigths reserved

```
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
...
```

Le dernier commentaire décrit la situation obtenue, dans laquelle il est clair que l'hypothèse **(H)** est redevenue vraie en ce sens qu'elle décrit, de nouveau, l'état courant dans lequel se trouve le système. Puisque le système se trouve à présent dans l'état décrit par **(H)**, on peut reprendre exactement les mêmes instructions qui vont opérer de la même façon sur les mêmes données. *On peut itérer, répéter les mêmes traitements.* On obtient donc :

```
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
si
    i - 1 = 10
alors
    c'est fini
sinon
      t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
      recommencer au début
fin si
```

Les traitements recommencent donc, identiques « mot pour mot » pour obtenir un nouvel état intermédiaire décrit par **(H)**, mais *cette répétition s'arrêtera-t-elle* ? À chaque étape, on initialise une nouvelle case et la valeur de `i` augmente à l'instruction d'incrémentation `i <- i+1`. Si cette valeur est initialement inférieure à 11 au début du calcul, elle finira par atteindre 11 et alors, l'expression booléenne `i - 1` = 10 prendra la valeur VRAI ce qui terminera l'itération. Si i n'est pas initialement inférieur ou égal à 11, sa valeur augmentera indéfiniment sans que celle de `i - 1` atteigne 10 et l'itération ne s'arrêtera pas. Il nous faut donc déterminer la valeur initiale de `i`.

Initialiser le calcul

Initialiser le calcul, c'est placer le système dans un état initial I qui rend vraie l'hypothèse **(H)**. Cette hypothèse dit que la dernière case déjà initialisée est celle de numéro `i - 1`. Par conséquent, la prochaine case à initialiser est celle de numéro `i`. Avant de commencer, aucune case n'a encore été initialisée et donc, la prochaine case à remplir est celle de numéro 0 :

```
initialisation
    i <- 0
```

L'hypothèse précise alors que `valeur` vaut `i x n`. Avec `i` = 0 pour valeur initiale de `i`, on obtient `valeur = 0 x n = 0` pour valeur initiale de `valeur` :

```
initialisation
    i <- 0
    valeur <- 0
```

Il reste à arranger l'écriture de cet algorithme. Commençons par la condition de terminaison. Nous avons écrit :

```
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
si
    i - 1 = 10
alors
    c'est fini
sinon
    ...
```

© ENI Editions - All rigths reserved

Nous pouvons écrire que `c'est finisi i-1=10` est équivalent à `c'est finisi i=11`. Autrement dit, c'est fini dès que i ≥ 11 puisque `i` est croissant. La négation de cette assertion est :

```
ce n'est pas fini tant que i < 11
```

Nous obtenons ainsi une première forme opérationnelle de l'algorithme cherché :

```
initialisation
    i <- 0
    valeur <- 0
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
tant que
    i < 11
répéter
      t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
fin répéter
```

La construction **tant que ... répéter ... fin répéter** répète les traitements imposés par les instructions entre **répéter** et **fin répéter**. Ces traitements ne sont jamais réalisés si la condition de contrôle du bloc **tant que** est fausse. Ils cessent dès que cette condition est évaluée à FAUX. Elle est évaluée avant la première exécution du bloc **répéter** puis à chaque exécution de **fin répéter**. Ce n'est pas la seule construction possible pour exprimer cette idée. En fait, elle a été obtenue par le raisonnement mené sur la condition d'arrêt de l'itération. Reprenons ce raisonnement autrement :

```
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
si
    i - 1 = 10
alors
    c'est fini
sinon
    ...
```

Nous pouvons écrire que `c'est finisi i-1=10` est équivalent à `c'est finisi i=11`. Autrement dit, c'est fini dès que i ≥ 11 puisque `i` est croissant. Par conséquent, les traitements sont à répéter **jusqu'à ce que** i ≥ 11 ou encore **jusqu'à ce que** i > 10. La forme opérationnelle de l'algorithme devient :

```
initialisation
    i <- 0
    valeur <- 0
# On a initialisé les cases de t jusqu'à la case de numéro i-1
# comprise. valeur = i x n
jusqu'à
    i > 10
répéter
      t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
fin répéter
```

Le « fonctionnement » de cette construction est rigoureusement identique à celui de la construction précédente. Le prédicat du bloc **jusqu'à** est évalué avant la première exécution du bloc **répéter** et à chaque exécution de **fin**

© ENI Editions - All rigths reserved

répéter. L'itération cesse dès que cette condition est évaluée à FAUX.

Il reste à terminer par la cinquième étape.

Rédiger l'algorithme définitif

À ce stade, nous avons un algorithme presque complet. La signature de la procédure, complétée de la précondition et de la postcondition, exprime clairement ce que fait l'algorithme. C'est la documentation communiquée à tout utilisateur de ses services. Un utilisateur n'a pas à connaître les détails internes de l'algorithme. La documentation est suffisante pour l'utiliser, c'est-à-dire pour savoir précisément l'état initial dans lequel doit se trouver le système logiciel client pour utiliser l'algorithme et l'état final obtenu après son exécution. Cependant, pour le programmeur, cela ne suffit pas. Il faut encore fournir, avec l'algorithme, les moyens pour en contrôler le fonctionnement. On doit pouvoir disposer de tout ce qui est nécessaire pour se convaincre qu'il est correct et même pour prouver qu'il est correct. Voyons cela en commençant par l'hypothèse **(H)**.

Hypothèse (H) : les cases de t jusqu'à la case de numéro i - 1, comprise, ont déjà été initialisées. La valeur dans la case de numéro i - 1 est (i - 1) x n. La variable `valeur` contient la prochaine valeur i x n à ranger.

Tout le raisonnement qui mène de cette hypothèse à l'algorithme obtenu consiste à faire en sorte qu'elle énonce une assertion qui est vraie dans l'état initial et qu'elle reste vraie dans tous les états intermédiaires. Quand on constate que des instructions placent le système dans un état intermédiaire dans lequel elle devient fausse, on « corrige » cet état pour qu'elle redevienne vraie. Dans l'étape intitulée « *Se rapprocher de l'état final* », on constate que, après avoir rempli une nouvelle case de t, **(H)** devient fausse et on corrige en incrémentant i. Mais alors, l'hypothèse n'est toujours pas vérifiée en ce qui concerne la valeur à placer dans la prochaine case à visiter et on la corrige en l'augmentant de n. Dans l'étape « *Initialiser le calcul* », on cherche à rendre **(H)** vraie à l'état initial. Par conséquent, cette hypothèse exprime une propriété qui doit rester vraie dans tous les états du système. C'est ce que l'on appelle une *propriété invariante* ou plus simplement un **invariant**. C'est un prédicat fondamental :

- il énonce une propriété permanente du système et maintenir cette propriété est nécessaire à la correction de l'algorithme ;
- c'est ce prédicat qui guide l'analyse de l'algorithme pour construire l'itération.

Nous verrons plus loin que la modification de cette hypothèse conduit à un algorithme dont l'expression est sensiblement différente de celle de l'algorithme qui vient d'être obtenu.

Revenons encore à **(H)**. Nous devons pouvoir exprimer cette hypothèse d'une façon plus formelle. Il existe plusieurs façons de le faire. Comme elle énonce une propriété en langage naturel, on souhaite d'abord l'exprimer mathématiquement. Dans ce cas, elle peut s'écrire, par exemple :

```
(∀k, 0 ≤ k ≤ i-1) (valeur = k x n et t[k] = valeur)
```

On exprime bien ainsi que toutes les cases, jusqu'à la case de numéro i – 1 comprise, ont été remplies avec la valeur k x n dans laquelle k est le numéro de la case. Cette expression est un élément important de la documentation interne de l'algorithme et participe à former notre conviction que l'algorithme est correct. On aurait pourtant apprécié d'exprimer cette hypothèse d'une façon plus « algorithmique », comme nous le faisons pour la précondition ou la postcondition. Il s'agit d'écrire que t contient les résultats de la table de multiplication par n dans toutes les cases dont les numéros sont strictement inférieurs à i . Nous disposons pour cela du prédicat **estTableDe** qui a été préparé exactement dans ce but. **(H)** s'écrit alors :

```
invariant
    i > 0 => estTableDe(t, 0, i-1, n)
```

L'expression de la propriété invariante suffit-elle à assurer la correction de l'algorithme ? La propriété étant vraie pour toute valeur de k entre 0 et i – 1, elle reste vraie si i n'augmente pas. Ainsi, l'expression précédente de **(H)** ne suffit pas pour se prémunir contre l'oubli de l'instruction qui incrémente i et cet oubli est grave, car alors, l'algorithme entre dans une itération infinie, qui ne se termine pas. Revenons donc à la condition d'arrêt de l'itération :

```
c'est fini dès que i - 1 = 10
```

avec initialement, i = 0

```
i - 1 = 10 ⇔ 10 - (i - 1) = 0 ⇔ 11 - i = 0
```

Ainsi, lorsque i augmente de 0 à 11, 11 – i décroît de 11 à 0. Nous avons là un moyen de vérifier que l'itération se terminera : s'assurer, *à chaque reprise de l'itération*, que la valeur 11 – i décroît strictement. Appelons **variant de contrôle** la quantité dont la valeur entière positive, dans une itération, décroît *strictement* vers 0. Remarquez que, pour ce problème, la valeur de 11 – i est, pour tout i , le nombre de cases de t qui ne sont pas encore initialisées.

© ENI Editions - All rigths reserved

On peut y penser comme à une mesure de la « distance » qui sépare l'état intermédiaire actuel de l'état final. Pour que l'itération soit correcte, il faut qu'elle se termine et elle se terminera si la distance à l'état final décroît strictement. C'est une condition nécessaire de correction, mais évidemment pas suffisante, comme cela a déjà été vu plus haut.

Finalement, l'algorithme devient :

Algorithme 3 : Table de multiplication - Version 2.1

```
Algorithme tableDe
    # Construire la table de multiplication par n.
Entrée
    n : ENTIER          # L'entier dont on veut la table.
    t : TABLEAU[ENTIER] # La table à initialiser.
précondition
    n ≥ 0
variable
    i      : ENTIER # Le numéro de la prochaine case à initialiser.
    valeur : ENTIER # La valeur d'initialisation de la case.
initialisation
    i <- 0
    valeur <- 0
jusqu'à
    i > 10
    invariant
        # On a initialisé les cases de t jusqu'à la case de numéro
        # i-1 comprise. valeur = i x n
        # (∀k, 0 ≤ k ≤ i-1) (valeur = k x n et t[k] = valeur)
        i > 0 => estTableDe(t, 0, i-1, n)
    variant de contrôle
        # Nombre de cases pas encore initialisées.
        11 - i
répéter
      t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
fin répéter
postcondition
    ancien(n) = n       # n n'est pas modifié.
    # (∀i, 0 ≤ i ≤ 10)(t[i] = i x n)
    estTableDe(t, 0, 10, n)
fin tableDe
```

La postcondition reprend, en commentaire, l'expression mathématique obtenue plus haut. Bien entendu, cet algorithme doit être complété par les spécifications de l'algorithme **estTableDe**.

Tout ce travail peut paraître long, compliqué et vain : est-il vraiment nécessaire de développer une telle analyse pour un petit problème comme la table de multiplication ? Évidemment non ! Tout apprenti programmeur sait produire, après quelques heures d'apprentissage et dans un langage procédural quelconque, un programme qui écrit à l'écran la table de multiplication par un entier quelconque. Cependant, il faut garder à l'esprit que l'objectif de cette section n'est pas d'obtenir la table de multiplication. Il s'agit de comprendre l'itération et une méthode d'analyse systématique pour construire d'une façon sûre des itérations. L'objectif ici est le raisonnement et son contrôle. Et pour bien insister sur ce point, étudions une autre version du même algorithme.

c. Une autre version

Nous voulons insister sur plusieurs points et, notamment, sur le fait que l'itération ne s'est pas construite n'importe comment. En particulier, c'est le calcul et un raisonnement rigoureux qui *imposent* les différentes composantes de l'algorithme. Reprenons l'hypothèse **(H)** de départ du raisonnement et modifions-la légèrement :

© ENI Editions - All rigths reserved

Hypothèse (H') : les cases de t jusqu'à la case de numéro i, comprise, ont déjà été initialisées. La valeur dans la case de numéro i est i x n. La variable `valeur` contient la prochaine valeur (i+1) x n à ranger.

Dans cette nouvelle hypothèse, i a été remplacé par i + 1 dans toutes ses occurrences. Reprenons une à une les étapes de la construction de l'itération à partir de cette nouvelle hypothèse.

Voir si c'est fini

C'est fini dès que i = 10, donc dès que i ≥ 10 puisque i est croissant. Par conséquent, ce n'est pas fini tant que i < 10.

Se rapprocher de l'état final

On avance vers la solution en initialisant la case suivante, celle de numéro i + 1. On obtient donc :

```
tant que
    i < 10
répéter
    i <- i + 1
    valeur <- valeur + n
    t[i] <- valeur
fin répéter
...
```

Initialiser le calcul

Pour réaliser les conditions que décrit l'hypothèse **(H')** dans l'état initial, on doit initialiser i de sorte que sa valeur soit le numéro de la dernière case remplie. Si i est le numéro de la dernière case remplie, alors i + 1 est le numéro de la prochaine case à remplir. Or, initialement, aucune case n'est encore remplie et, par conséquent, le numéro de la prochaine case à remplir est **index_min**(t). Ainsi, initialement :

```
i + 1 = 0 ⇔ i = 0 - 1 = -1
```

De même :

```
valeur = i x n ⇔ valeur = -1 x n = -n
```

Les initialisations deviennent :

```
initialisation
    i <- -1      # Numéro dernière case initialisée.
    valeur <- -n # Valeur à initialiser.
```

Rédiger l'algorithme définitif

Pour rédiger l'algorithme, il reste à exprimer l'invariant et le variant de contrôle. L'invariant est déduit directement de ce que i est le numéro de la dernière case remplie, dès que i ≥ 0 :

```
invariant
    i ≥ 0 ⇒ estTableDe(t, 0, i, n)
```

Le variant de contrôle est déduit du prédicat de contrôle de l'itération. Celle-ci termine lorsque i = 10 et donc lorsque 10 – i = 0 par valeur décroissante. Par conséquent, le variant de contrôle est 10 – i :

```
variant de contrôle
    10 - i
```

Il devient possible, à présent, de rédiger l'algorithme définitif :

Algorithme 4 : Table de multiplication - Version 2.2

```
Algorithme tableDe
    # Construire la table de multiplication par n.
```

© ENI Editions - All rigths reserved

```
Entrée
    n : ENTIER          # L'entier dont on veut la table.
    t : TABLEAU[ENTIER] # La table à initialiser.
précondition
    n ≥ 0
variable
    i      : ENTIER # Le numéro de la dernière case initialisée.
    valeur : ENTIER # La valeur d'initialisation de la case.
initialisation
    i <- -1
    valeur <- -n
tant que
    i < 10
    invariant
        # On a initialisé les cases de t jusqu'à la case de numéro i
        # comprise. valeur = i x n
        # (∀k, 0 ≤ k ≤ i-1) (valeur = k x n et t[k] = valeur)
        i ≥ 0 => estTableDe(t, 0, i, n)
    variant de contrôle
        # Nombre de cases pas encore initialisées.
        10 - i
répéter
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
      i <- i+1
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = (i-1) x n
      valeur <- valeur + n
    # On a initialisé les cases de t jusqu'à la case de numéro i-1
    # comprise. valeur = i x n
      t[i] <- valeur
    # On a initialisé les cases de t jusqu'à la case de numéro i
    # comprise. valeur = i x n
fin répéter
postcondition
    ancien(n) = n       # n n'est pas modifié.
    # (∀i, 0 ≤ i ≤ 10)(t[i] = i x n)
    estTableDe(t, 0, 10, n)
fin tableDe
```

Ce n'est plus le même algorithme que celui de la version 2.1. La modification de l'hypothèse, derrière laquelle se cache la propriété invariante de l'itération, a imposé :

- de nouvelles initialisations pour placer le système dans l'état initial :
  - i = -1 au lieu de i = 0 ;
  - valeur = -n au lieu de valeur = 0 ;
- une nouvelle condition de fin d'itération : i < 10 au lieu de i < 11 ;
- une nouvelle position pour les différentes instructions qui rétablissent l'invariant : nouvelles valeurs de i et de la variable valeur ;
- l'expression d'un nouvel invariant et d'un nouveau variant de contrôle.

Encore une fois, toutes ces modifications ne sont pas obtenues par l'intuition et le bricolage, mais par un raisonnement rigoureux. C'est l'enseignement essentiel de ce livre. Bien entendu, comme dans toute activité humaine, nous restons à la merci d'une erreur de raisonnement ou d'inattention. Peut-être subsiste-t-il d'ailleurs de telles erreurs dans ce qui précède. L'algorithmique est une activité difficile. Raison de plus pour mettre en œuvre des méthodes de raisonnement qui aident à construire des algorithmes sûrs. Cette façon de procéder vaut infiniment mieux que ces « bidouillages » infinis devant un clavier d'ordinateur, à la recherche désespérée de la solution « qui marche » et à laquelle on ne pourra jamais faire confiance, sauf peut-être pour des problèmes triviaux. Il en est de même des notations. Nous y reviendrons.

© ENI Editions - All rigths reserved

Exercice 1 : Versions itératives d'algorithmes récursifs

Cet exercice propose de reprendre la définition d'algorithmes étudiés au chapitre Récursivité.

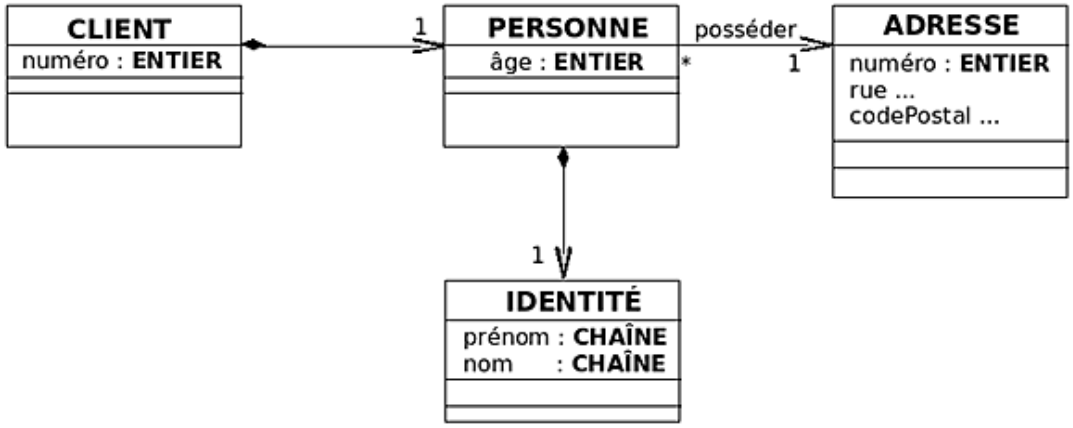
*1. Écrire une version itérative de la fonction **rang** définie au chapitre précédent.*

*2. Même question pour la fonction **longueur** qui calcule le nombre de caractères d'une chaîne.*

Les sections suivantes étudient complètement d'autres exemples.

2. Explorer un tableau

Un **CLIENT** est une **PERSONNE** identifiée par un numéro entier. Une **PERSONNE** possède une **ADRESSE** et une **IDENTITÉ**. Le diagramme ci-dessous représente ces entités et leurs associations mutuelles.



Remarquez la nature particulière de l'association entre **CLIENT** et **PERSONNE** ou entre **PERSONNE** et **IDENTITÉ**. On indique ainsi qu'une instance de **CLIENT** est composée d'une et d'une seule instance de **PERSONNE**. En toute rigueur, il faudrait utiliser ici l'association d'héritage, mais nous ne sommes pas en algorithmique objet. De plus, la couleur du losange du côté **CLIENT** indique que, lorsque le client disparaît, la personne disparaît avec lui.

Le type **CLIENT** est défini par la déclaration :

```
type
    CLIENT
structure
    numéro : ENTIER
    p      : PERSONNE
fin CLIENT
```

La déclaration du type **PERSONNE** est de même :

```
type
    PERSONNE
structure
    identité : IDENTITÉ
    âge      : ENTIER
    adresse  : ADRESSE
fin PERSONNE
```

Enfin, le type **ADRESSE** a déjà été utilisé au chapitre Structures élémentaires.

Un tableau, qui peut contenir jusqu'à 1000 **CLIENTs** a été déclaré et initialisé :

```
constante
    MAX_CLIENTS : ENTIER <- 1000
    MIN_CLIENTS : ENTIER <- 1
    VIDE   : ENTIER <- ??? # Numéro d'un client qui n'existe pas.
    EFFACÉ : ENTIER <- ??? # Numéro d'un client effacé.
...
variable
```

© ENI Editions - All rigths reserved

```
    clients : TABLEAU[CLIENT][MIN_CLIENTS, MAX_CLIENTS]
...
```

Actuellement, toutes les cases n'ont pas nécessairement été initialisées par des clients. L'entreprise n'a peut-être pas encore 1000 clients. La première case qui n'a pas encore été initialisée porte un numéro de client initialisé à la valeur de la constante VIDE. C'est une constante quelconque, mais qui ne peut être un numéro de client valide. Dans la vie de l'entreprise, un client « naît, vit et meurt ». Autrement dit, de nouveaux clients sont inscrits, d'autres sont effacés du tableau. Lorsque l'un d'eux est effacé, on se contente d'indiquer que son numéro prend la valeur EFFACÉ. Voici, par exemple, comment il est possible de définir ces valeurs particulières :

```
constante
    MAX_CLIENTS : ENTIER <- 1000
    MIN_CLIENTS : ENTIER <- 1
    # Numéro d'un client qui n'existe pas.
    VIDE    : ENTIER <- MIN_CLIENTS - 1
    # Numéro d'un client effacé.
    EFFACÉ : ENTIER <- VIDE - 1
```

a. Rechercher une identité : le problème

On donne l'identité d'un client particulier :

```
variable
    id : IDENTITÉ
...
id.prénom <- 'Jacques'
id.nom    <- 'MARTIN'
...
```

Remarquez comment est utilisée une donnée du type **IDENTITÉ**. Une instance de ce type est faite de deux chaînes de caractères : un prénom et un nom. L'opérateur *point* ('.') est utilisé pour sélectionner un champ particulier, un attribut, de l'instance. Ainsi, pour initialiser le prénom de l'identité i d, on écrit :

```
id.prénom <- 'Jacques'
```

Faire un algorithme qui détermine le numéro de la case du tableau des clients qui contient les données de ce client particulier.

Il s'agit donc de parcourir le tableau des clients pour y chercher la position de l'un d'eux dont on donne l'identité. On veut connaître le numéro de la case qui le contient. Le tableau ci-dessous représente un exemple d'un tel tableau.

1	212	Jacques	MARTIN	25	17	Square des lilas	75210	OISÉ
2	36	Raimond	ZITTON	32	...	...	...	...
3	-2	Alain	DUPONT	47	...	...	...	...
...	...	...	...	...	...	...	...	...
28	76	Jacques	MARTIN	33	26	Rue des bois	...	...
...	...	...	...	...	...	...	...	...
85	177	Marcel	DUCHAMP	56	1	Allée des oiseaux	...	...
86	-1	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
1000	-1	...	...	...	...	...	...	...

Les pointillés remplacent soit des données non initialisées, comme en dernière ligne par exemple, ou des valeurs qui ne sont pas précisées, comme dans la deuxième ligne par exemple.

© ENI Editions - All rigths reserved

La première colonne précise le numéro de la case du tableau. C'est MIN_CLIENTS pour la première case et MAX_CLIENTS pour la dernière. Ici, ces valeurs sont respectivement 1 et 1000. Les autres colonnes contiennent les valeurs des attributs de clients enregistrés. Remarquez qu'il existe plusieurs clients dont l'identité est la même. Cependant, le numéro est un identifiant qui ne peut apparaître qu'en un seul exemplaire. La définition de l'algorithme doit préciser quel rang il renvoie dans ce cas. Convenons que, dans le cas d'identités multiples, on attend le numéro de case le plus petit, autrement dit le numéro de la première case contenant l'identité cherchée. Ici, ce serait 1 pour 'Jacques MARTIN'. Dans ce tableau, 'Alain DUPONT' a été effacé : le numéro de client vaut -2. Les cases entre les numéros 86 et 1000 sont vides : elles ont été initialisées par un numéro de client qui vaut -1.

Pour trouver une identité donnée, l'algorithme observe le contenu de la première case du tableau `clients` et regarde d'abord si le client existe. Si c'est le cas, il compare l'identité enregistrée à celle cherchée. Si la comparaison réussit, il renvoie 1 ; sinon, il observe le contenu de la deuxième case et ainsi de suite. La recherche se termine, soit parce que l'identité cherchée est trouvée et alors l'algorithme rend le numéro de la case qui la contient, soit parce le numéro de client dans cette case a la valeur VIDE lorsque tout le tableau initialisé a été parcouru sans trouver d'instance de l'identité cherchée. Dans ce cas, il faut convenir d'une valeur que retournera l'algorithme. Soit ABSENT cette valeur.

```
constante
    MAX_CLIENTS : ENTIER <- 1000
    MIN_CLIENTS : ENTIER <- 1
    # Numéro d'un client qui n'existe pas.
    VIDE    : ENTIER <- MIN_CLIENTS - 1
    # Numéro d'un client effacé.
    EFFACÉ : ENTIER <- VIDE - 1
    # Résultat d'une recherche qui échoue.
    ABSENT : ENTIER <- VIDE - 2
```

b. L'algorithme à écrire : chercherIdentité

Soit **chercherIdentité**, l'algorithme à écrire. Comme nous en avons l'habitude à présent, sa responsabilité sera de chercher une identité dans les cases du tableau clients situées entre deux cases extrêmes dont on donne les numéros. Il est possible de spécifier cet algorithme. Ainsi, il deviendra possible de l'utiliser sans avoir besoin d'en connaître les détails de réalisation.

*Algorithme 5 : Spécifications de **chercherIdentité***

```
Algorithme chercherIdentité
    # Le numéro de la case de t entre début et fin qui contient
    # identité ou ABSENT sinon.
Entrée
    t : TABLEAU[CLIENT]
    début, fin : ENTIER
    identité   : IDENTITÉ
Résultat : ENTIER
précondition
    t ≠ NUL
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    début > fin => Résultat = ABSENT
    début ≤ Résultat ≤ fin ⇒
        (
            t[Résultat].p.identité = identité
          et
            t[Résultat].numéro ≠ EFFACÉ
          et
            t[Résultat].numéro ≠ VIDE
        )
    # Rend ABSENT si identité n'est pas dans t[début .. fin].
    non estDans(t, début, fin, identité) ⇒ Résultat = ABSENT
    # Rend la première valeur rencontrée.
    début < Résultat ⇒ non estDans(t, début, Résultat-1, identité)
fin chercherIdentité
```

Remarquez, là encore, l'accès aux champs d'une instance de type structuré. Le tableau t contient des **CLIENTs** dans ses cases. Pour sélectionner le client enregistré dans la case de numéro 5 par exemple, on utilise la notation t[5]. Un client est fait d'un numéro et d'une personne désignée par p dans la définition de **CLIENT**. Pour accéder aux

© ENI Editions - All rigths reserved

éléments qui définissent la personne associée au numéro 5, on utilise la notation t[5].p. Puis, pour accéder à l'identité de cette personne, on utilise t[5].p.identité puisque cette identité est désignée par identité dans la définition de **PERSONNE**. Enfin, quand on aura besoin d'accéder, par exemple, au nom du client, on utilisera la notation t[5].p.identité.nom

Remarquez aussi comment sont comparées deux identités. La postcondition fait appel au prédicat t [**Résultat**].p.identité = identité qui utilise le signe d'égalité habituel pour s'assurer que les deux identités sont égales. Il n'y a pas de précaution particulière à prendre ici puisque nous sommes en algorithmique et que la notation est claire. En programmation, il faudrait s'assurer que l'opérateur '=' est bien défini sur les types de données concernés.

Remarquez enfin les différentes clauses de la postcondition. La première s'assure que début ≤ fin et que la fonction rend ABSENT lorsque ce n'est pas le cas. La deuxième dit que le résultat entre début et fin est un numéro de case qui contient l'adresse cherchée et que le client concerné existe, c'est-à-dire n'a pas été effacé et que la case correspondante n'est pas vide. Pour une identité qui n'est pas présente dans le tableau, le résultat est ABSENT. Enfin, lorsque le résultat est un numéro de case strictement supérieur à début, aucune case de numéro compris entre début et **Résultat**-1 ne contient un client ayant l'identité cherchée. On précise bien ainsi que la fonction retourne le numéro de la case qui contient la première occurrence rencontrée pour cette identité.

Le prédicat **estDans** rend VRAI si et seulement si le paramètre identité est dans une case de t de numéro compris entre début et fin. Ses spécifications sont :

*Algorithme 6 : Spécifications de **estDans** pour une identité*

```
Algorithme estDans
    # identité est-elle dans t, entre les cases de numéros début et
    # fin ?
Entrée
    t : TABLEAU[CLIENT]
    début, fin : ENTIER
    identité   : IDENTITÉ
Résultat : BOOLÉEN
précondition
    t ≠ NUL
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    début > fin => Résultat = FAUX
    début = fin => Résultat =
        (
            t[début].p.identité = identité
          et
            t[début].numéro ≠ EFFACÉ
          et
            t[début].numéro ≠ VIDE
        )
    début < fin ⇒ Résultat =
        (
            (
                t[début].p.identité = identité
              et
                t[début].numéro ≠ EFFACÉ
              et
                t[début].numéro ≠ VIDE
            )
          ou sinon
            estDans(t, début+1, fin identité)
        )
fin estDans
```

On cherche une identité donnée en écrivant, par exemple :

```
variable
    ...
    id   : IDENTITÉ
    rang : ENTIER
...
id.prénom <- 'Jacques'
```


© ENI Editions - All rigths reserved

```
id.nom <- 'MARTIN'
...
rang <- chercherIdentité(clients, MIN_CLIENTS, MAX_CLIENTS, id)
si
    rang ≠ ABSENT
alors
    'Jacques MARTIN' trouvé dans la case de numéro rang.
sinon
    Pas de 'Jacques MARTIN' parmi les clients enregistrés.
fin si
...
```

Supposons d'abord que **chercherIdentité** fait partie de notre bibliothèque de composants logiciels et utilisons-la pour résoudre quelques problèmes simples où intervient l'itération.

c. Utilisation : effacer tous les clients d'identité donnée

Exercice résolu 1 : Effacer les clients dont on donne l'identité

On donne une identité id.

Effacer tous les clients ayant cette identité.

Solution

Nous avons vu comment procéder plus haut. On observe une case et on passe à la suivante en cas d'échec. Dans un état intermédiaire de la recherche, on a déjà parcouru et traité quelques cases du tableau. Reprenons les étapes de construction d'une itération.

Faire une hypothèse sur l'état actuel

Hypothèse (H) : l'identité id a été cherchée dans clients. Le dernier numéro de case reçu en résultat et non encore traité est rang.

La variable rang contient donc un numéro de case et ce numéro de case n'a pas encore été traité. Autrement dit, si la case de numéro rang contient un client valide, ce client n'est pas encore effacé.

Voir si c'est fini

C'est fini lorsque rang = ABSENT qui est la valeur convenue rendue par **chercherIdentité**, quand il ne trouve pas de réponse. Lorsque rang ≠ ABSENT, on a dans rang le numéro d'une case du tableau clients qui contient un client valide dont l'identité est celle cherchée. Il est alors possible de traiter ce client.

```
si
    rang = ABSENT
alors
    c'est fini
sinon
    traiter le client de la case rang.
...
```

Se rapprocher de l'état final

Pour se rapprocher de l'état final, on doit effacer ce client. On obtient alors :

```
...
sinon
    # clients[rang] contient un client à effacer.
    effacer le client dans la case rang
    calculer le rang du client suivant de même id
...
```

L'opération effacer le client dans la case rang consiste à placer la valeur de la constante EFFACÉ dans le

© ENI Editions - All rigths reserved

numéro de client de la case de numéro rang :

```
...
    clients[rang].numéro <- EFFACÉ
    # rang est le dernier numéro de case reçu et traité.
...
```

L'état décrit par le dernier commentaire n'est pas un état intermédiaire décrit par **(H)**. Pour obtenir un tel état, il suffit de calculer le rang suivant dans le tableau clients. Mais alors que la première recherche se fait à partir de la première case de clients, il faut à présent chercher le client suivant, donc un client de numéro supérieur à rang :

```
...
    # rang est le dernier numéro de case reçu et traité.
    rang <- chercherIdentité(clients, rang+1, MAX_CLIENTS, id)
    # rang est le dernier numéro de case reçu et non encore traité.
...
```

On retrouve alors **(H)** et il devient possible d'*itérer*, de refaire les calculs. On sait que l'algorithme termine parce que le numéro de la case où commence une nouvelle recherche est strictement supérieur au numéro de la case résultat de l'itération précédente. Chaque appel nous rapproche de la fin du tableau et on finira par atteindre la dernière case valide. L'algorithme rendra alors ABSENT qui terminera l'itération. Bien entendu, ceci suppose que **chercherIdentité** est correct, mais ce n'est pas actuellement notre responsabilité d'assurer la correction de cette fonction. On se conforme à sa précondition et on reçoit le service attendu si la postcondition nous satisfait ; sinon, il faut aller voir ailleurs.

Initialiser le calcul

Initialiser le calcul, c'est placer le système logiciel dans un état dans lequel **(H)** est réalisée. Il suffit donc de faire appel à **chercherIdentité** pour obtenir une première valeur de rang :

```
...
    rang <- chercherIdentité(clients, MIN_CLIENTS, MAX_CLIENTS, id)
    # rang est le dernier numéro de case reçu et non encore traité.
...
```

Rédiger l'algorithme définitif

Commençons par arranger la condition de terminaison de l'algorithme. C'est fini lorsque rang = ABSENT et, par conséquent, ce n'est pas fini tant que rang ≠ ABSENT. Comme rang contient le numéro de la prochaine case à traiter lorsque ce n'est pas fini, il ne reste plus qu'à rédiger la version définitive.

L'invariant est plus difficile à obtenir. Dans un état intermédiaire, **(H)** précise que les clients dont l'identité est id ont été effacés dans toutes les positions qui précèdent rang. Soit, alors le prédicat **estEffacé** dont la signature est :

```
estEffacé
    (
      t : TABLEAU[CLIENT]    # Tableau à traiter.
      début, fin : ENTIER    # Numéros des case extrêmes concernées.
      identité   : IDENTITÉ  # Identité à effacer.
    ) : BOOLÉEN
    # Les clients d'identité identité enregistrés dans
    # t[début .. fin] ont-ils été effacés ?
```

La précondition est classique. La postcondition exprime que tous les clients enregistrés dans les cases entre t[début] et t[fin] d'identité identité ont été effacés, mais aussi qu'aucun autre client n'a été modifié. Pour s'assurer qu'aucun autre client n'a été modifié, il faut pouvoir comparer l'enregistrement qui correspond à un client donné avant et après effacement. Or, une fonction ne peut pas utiliser la construction **ancien** sur l'un de ses paramètres : pour une fonction, la valeur d'un paramètre en entrée reste égale à sa valeur en sortie puisqu'il s'agit d'une requête, qui ne modifie pas l'état du système. Par conséquent, on modifie la signature du prédicat **estEffacé** pour qu'il assure aussi la vérification de l'invariance des enregistrements du tableau t dans les positions qui ne correspondent pas à l'identité cherchée. La procédure qui efface utilisera le prédicat ainsi :

```
...
estEffacé(ancien(clients), clients, MIN_CLIENTS, MAX_CLIENTS, id)
...
```

Ce prédicat retourne VRAI si et seulement si tous les clients d'identité id enregistrés dans **ancien**(clients), entre les composantes de numéros MIN_CLIENTS et MAX_CLIENTS, ont été effacés **et** aucune autre composante du tableau

© ENI Editions - All rigths reserved

n'a été modifiée. L'écriture de la précondition étant laissée en exercice, on obtient l'algorithme suivant :

*Algorithme 7 : Spécifications de **estEffacé** pour une identité donnée*

```
Algorithme estEffacé
    # Les clients d'identité identité enregistrés dans
    # t[début .. fin] ont-ils été effacés ?
Entrée
    at, t : TABLEAU[CLIENT] # Tableaux à traiter.
    début, fin : ENTIER     # Numéros des case extrêmes concernées.
    identité   : IDENTITÉ   # Identité à effacer.
Résultat : BOOLÉEN
précondition
    ... laissée en exercice ...
postcondition
    début > fin => Résultat = FAUX
    début = fin => Résultat =
        (
            (
                at[début].p.identité = identité
              et
                at[début].numéro ≠ EFFACÉ
              et
                at[début].numéro ≠ VIDE
              et
                t[début].numéro =  EFFACÉ
              et
                at[début]. p.identité = t[début].p.identité
            )
          ou sinon
            (
                at[début].p.identité ≠ identité
              et
                at[début] = t[début]
            )
        )
    début < fin => Résultat =
        (
            (
                at[début].p.identité = identité
              et
                at[début].numéro ≠ EFFACÉ
              et
                at[début].numéro ≠ VIDE
              et
                t[début].numéro =  EFFACÉ
              et
                at[début]. p.identité = t[début].p.identité
            )
          ou sinon
            (
                at[début].p.identité ≠ identité
              et
                at[début] = t[début]
            )
          et alors
            estEffacé(at, t, début+1, fin, identité)
        )
fin estEffacé
```

Revenons alors à la procédure qui efface. Le prédicat **estEffacé** permet d'exprimer l'invariant de l'itération :

```
invariant
    rang > MIN_CLIENTS =>
        estEffacé(ancien(clients), clients, MIN_CLIENTS, rang-1, id)
```

© ENI Editions - All rigths reserved

Dans un état intermédiaire quelconque, la distance à la solution est majorée par le nombre de cases qui reste à explorer. C'est :

```
variant de contrôle
    MAX_CLIENTS - rang + 1
```

D'où la version complète de l'algorithme qui efface les clients.

Algorithme 8 : Effacer les clients d'identité donnée

```
constante
    MAX_CLIENTS : ENTIER <- 1000
    MIN_CLIENTS : ENTIER <- 1
    VIDE    : ENTIER <- index_min(clients) - 1
    EFFACÉ : ENTIER <- VIDE - 1
    ABSENT : ENTIER <- VIDE - 2
variable
    clients : TABLEAU[CLIENT][MIN_CLIENTS, MAX_CLIENTS]
    id      : CHAÎNE # L'identité cible des effacements.
...
id <- (l'identité à chercher pour effacement)
# Le tableau contient déjà les données à traiter. Effacer tous les
# clients d'identité donnée.
rang <- chercherIdentité(clients, MIN_CLIENTS, MAX_CLIENTS, id)
# Le tableau clients a déjà été parcouru à la recherche de id.
# rang est le dernier numéro de case obtenu en résultat et pas
# encore traité.
invariant
    rang > MIN_CLIENTS => estEffacé
            (ancien(clients), clients, MIN_CLIENTS, rang-1, id)
variant de contrôle
    MAX_CLIENTS - rang + 1
tant que
    rang ≠ ABSENT
répéter
    clients[rang].numéro <- EFFACÉ
    # Le tableau clients  a déjà été parcouru à la recherche de id.
    # rang est le dernier numéro de case obtenu en résultat est
    # traité.
    rang <- chercherIdentité(clients, rang + 1, MAX_CLIENTS, id)
fin répéter
```

Ayant ainsi étudié comment utiliser la fonction **chercherIdentité**, voyons à présent comment la définir.

d. Définition de chercherIdentité

La fonction implémente une itération qui permet d'explorer le tableau case par case. Dès qu'elle rencontre une case contenant l'identité cherchée, elle retourne le numéro de cette case, à ne pas confondre avec le numéro de client. L'analyse suit encore les étapes introduites précédemment.

Faire une hypothèse sur l'état actuel

On suppose que le tableau a déjà été partiellement exploré. Si on cherche encore, c'est que l'identité cherchée n'a pas encore été trouvée.

Hypothèse (H) : le tableau a été observé jusqu'à la case numéro i - 1 incluse et rien n'a été trouvé.

Voir si c'est fini

C'est fini et alors l'identité cherchée n'est pas trouvée dès que i - 1 = fin.

Se rapprocher de l'état final

© ENI Editions - All rigths reserved

On se rapproche de l'état final en observant une case de plus, donc en observant la prochaine case, celle de numéro i. Cependant, cette case ne contient pas nécessairement une donnée valide. Ainsi, l'identité quelle contient peut être celle cherchée alors que le client concerné a déjà été effacé lors d'une opération antérieure. De même, la case peut être vide si elle n'a jamais été initialisée avec les caractéristiques d'un client. Il s'agit de discriminer ces cas :

```
si
    table[i].numéro = VIDE
alors
    c'est fini : toute la table est explorée et l'identité n'est pas
    trouvée.
sinon si
    table[i].numéro= EFFACÉ
alors
    # observer la case suivante.
    i <- i + 1
sinon si
    table[i].p.identité = identité
alors
    c'est fini : identité cherchée trouvée en i.
sinon
...
```

Sinon, une case de plus vient d'être observée. L'état intermédiaire atteint se caractérise par la propriété suivante :

```
...
sinon
    # le tableau a été observé jusqu'à la case numéro i incluse
    # et rien n'a été trouvé.
```

Ce n'est pas tout à fait **(H)**. On la retrouve en avançant la valeur de i :

```
    i <- i + 1
    # le tableau a été observé jusqu'à la case numéro i - 1 incluse
    # et rien n'a été trouvé.
```

Il devient alors possible de *recommencer* les mêmes traitements. Il reste à démarrer le calcul. C'est fait en réalisant l'état initial dans lequel **(H)** est vraie lorsqu'aucune case n'a encore été observée :

Initialiser le calcul

La prochaine case à observer est celle de numéro i d'après **(H)**. Initialement, c'est celle de numéro début. Par conséquent, i est initialisé par :

```
initialisation
    i <- début
    ...
```

D'autre part, initialement, rien n'a encore été trouvé et différentes circonstances terminent le calcul. Soit alors fini, une variable booléenne vraie si et seulement si l'itération est terminée :

```
variable
    fini : BOOLÉEN
    ...
initialisation
    i    <- début
    fini <- FAUX
    ...
```

Cette variable prend la valeur VRAI lorsque le tableau est épuisé ou que l'identité cherchée est trouvée et, dans ce cas, **Résultat** prend une valeur différente de ABSENT. On peut alors rédiger l'algorithme définitif.

Rédiger l'algorithme définitif

C'est fini dès que fini prend la valeur VRAI. C'est le cas lorsque i > fin ou sinon lorsque table[i].numéro = VIDE ou sinon lorsque **Résultat** ≠ ABSENT. La variable fini devient donc inutile :

```
fini = i > fin ou sinon table[i].numéro = VIDE
```

© ENI Editions - All rigths reserved

```
            ou sinon Résultat ≠ ABSENT
```

D'autre part, l'invariant exprime **(H)** :

```
invariant
    i > début => non estDans(table, début, i - 1, identité)
    ...
```

Enfin, le nombre de cases qui n'ont pas encore été explorées est :

```
variant de contrôle
    fin - i + 1
```

D'où l'algorithme :

*Algorithme 9 : Définition de **chercherIdentité***

```
Algorithme chercherIdentité
    # Le numéro de la case de t[début .. fin] qui contient identité
    # ou sinon ABSENT.
Entrée
    t : TABLEAU[CLIENT] # Le tableau à explorer.
    Début, fin : ENTIER # Numéros cases extrêmes de l'exploration.
    identité   : IDENTITÉ # Identité cherchée.
    Résultat : BOOLÉEN
précondition
    ...
variable
    i : ENTIER # Numéro de la prochaine case à observer.
initialisation
    i <- début
    Résultat <- ABSENT
jusqu'à
      i > fin
    ou sinon
      table[i].numéro = VIDE
    ou sinon
      Résultat = ABSENT
invariant
    # (∀k, début ≤ k ≤ i - 1) (t[k].p.identité ≠ identité)
    i > début => non estDans(t, début, i - 1, identité)
variant de contrôle
    fin - i + 1
répéter
    si
        t[i].numéro ≠ EFFACÉ
      et alors
        t[i].p.identité = identité
    alors
        Résultat <- i
    sinon
        i <- i + 1
    fin si
fin répéter
postcondition
    ...
fin chercherIdentité
```

Peut-on écrire :

```
...
si
    t[i].p.identité = identité
alors
    Résultat <- i
```


© ENI Editions - All rigths reserved

```
fin si
i <- i + 1
...
```

Dans cette solution, on considère que, quoi qu'il en soit du résultat du test sur l'identité de la case actuelle, on peut toujours incrémenter i . Si l'identité a été trouvée, l'incrémentation de i ne sert à rien, mais comme l'itération se termine, cette opération n'a pas de conséquence ; sinon, il faut incrémenter i et c'est ce que fait cette instruction.

Cette solution est fausse, mais elle donne un résultat correct ! Elle est fausse parce que, dans le cas où le résultat est dans la case de numéro i , incrémenter i rend faux l'invariant : il n'est plus VRAI que **non estDans**(…) pour toutes les cases, jusqu'à la case de numéro i – 1 puisque, après l'incrémentation, le résultat est dans la case i – 1. Ce n'est donc pas une « bonne solution ».

e. Vieillir les clients

L'âge d'un client est enregistré dans le tableau. On gagne ainsi en calcul : enregistrer la date de naissance impose de recalculer l'âge chaque fois qu'il est nécessaire à un traitement, par exemple pour une campagne de publicité ciblée sur une tranche d'âges. En contrepartie, enregistrer l'âge au lieu de la date de naissance impose une mise à jour de cette propriété à chaque changement d'année. On suppose que l'année vient de changer. *Faire un algorithme qui ajuste l'âge des clients.*

C'est encore un algorithme de parcours du tableau des clients, mais cette fois, toutes les cases doivent être explorées et, éventuellement, modifiées. On visite chaque case et, pour chacune, on ajoute 1 à p.âge si elle ne contient ni VIDE ni EFFACÉ.

Faire une hypothèse sur l'état actuel

Dans un état intermédiaire du parcours, des cases du tableau ont été visitées. Disons que la dernière case visitée est celle de numéro i + 1 pour changer.

Hypothèse (H) : les âges ont été ajustés dans toutes les cases, jusqu'à celle de numéro i + 1 inclus.

Voir si c'est fini

C'est fini lorsque la prochaine case à visiter est une case qui contient VIDE pour le numéro du client, ou lorsque la dernière case visitée est celle de numéro **index_max**(clients). Il est donc possible d'écrire, provisoirement :

```
# les âges ont été ajustés dans toutes les cases, jusqu'à celle de
# numéro i + 1 inclus.
si
      i + 1 = MAX_CLIENTS
    ou sinon
      clients[i + 2].numéro = VIDE
alors
    c'est fini
sinon
...
```

Remarquez que, ici, la construction **ou sinon** s'impose. Il n'est pas possible d'utiliser **ou** seul. En effet, nous devons supposer que **ou** évalue les deux membres du prédicat. Or, lorsque i + 1 = MAX_CLIENTS, l'accès à la case de numéro i + 2 pour vérifier qu'elle n'est pas vide est illégal puisque, dans ce cas, cette case n'existe pas. La construction **ou sinon** est telle que son deuxième membre n'est évalué que lorsque son premier membre prend la valeur FAUX.

Sinon, il faut se rapprocher de la solution.

Se rapprocher de l'état final

Il s'agit alors de traiter la case suivante, de numéro i + 2. Cependant, cette case peut contenir un client qui a été effacé :

```
sinon
    si
        clients[i+2].numéro ≠ EFFACÉ
    alors
        # Vieillir ce client.
        clients[i + 2].p.âge <- clients[i + 2].p.âge + 1
```

© ENI Editions - All rigths reserved

```
    sinon
        # ce client est effacé : ne rien faire.
        Rien
    fin si
    # les âges ont été ajustés dans toutes les cases, jusqu'à celle
    # de numéro i + 2 inclus.
    i <- i + 1
    # les âges ont été ajustés dans toutes les cases, jusqu'à celle
    # de numéro i + 1 inclus.
    Recommencer
fin si
...
```

Initialiser le calcul

La dernière case visitée est celle de numéro i + 1 d'après **(H)**. La prochaine est celle de numéro i + 2. Initialement, la prochaine case à visiter est celle de numéro MIN_CLIENTS. Par conséquent, la valeur initiale de i est telle que i + 2 = MIN_CLIENTS, soit i = MIN_CLIENTS – 2.

```
initialisation
    i <- MIN_CLIENTS – 2
```

Il reste à rédiger l'algorithme.

Rédiger l'algorithme définitif

Arrangeons d'abord la condition de terminaison. C'est fini lorsque :

```
    i + 1 = MAX_CLIENTS ou sinon clients[i+2].numéro = VIDE
  ⇔
    i = MAX_CLIENTS – 1 ou sinon clients[i+2].numéro = VIDE
```

donc dès que :

```
    i ≥ MAX_CLIENTS – 1 ou sinon clients[i+2].numéro = VIDE
  ⇔
    i > MAX_CLIENTS – 2 ou sinon clients[i+2].numéro = VIDE
```

D'où une première version de l'algorithme cherché :

Algorithme 10 : Vieillir tous les clients de 1 an - Version 1.0

```
Algorithme vieillir
    # Augmenter l'âge de tous les clients de 1 an.
Entrée
    clients : TABLEAU[CLIENT]
précondition
    # Le tableau a été initialisé.
    (∀i, MIN_CLIENTS ≤ i ≤ MAX_CLIENTS)(clients[i] ≠ NUL)
constante
    MIN_CLIENTS      <- index_min(clients)
    VIDE    : ENTIER <- MIN_CLIENTS - 1
    EFFACÉ  : ENTIER <- MIN_CLIENTS - 2
    ABSENT  : ENTIER <- MIN_CLIENTS - 3
variable
    i : ENTIER # Numéro de la prochaine case à traiter.
initialisation
    i <- MIN_CLIENTS – 2
jusqu'à
    i > MAX_CLIENTS – 2 ou sinon clients[i+2].numéro = VIDE
répéter
    si
        clients[i+2].numéro ≠ EFFACÉ
    alors
        # Vieillir ce client.
```


© ENI Editions - All rigths reserved

```
        clients[i+2].p.âge <- clients[i+2].p.âge + 1
    sinon
        # Ce client est effacé : ne rien faire.
        Rien
    fin si
    i <- i + 1
fin répéter
postcondition
    # Ne modifie pas une case VIDE ou EFFACÉE.
    (∀i, MIN_CLIENTS ≤ i ≤ MAX_CLIENTS)
    (
        ancien(clients)[i].numéro = VIDE
      ou
        ancien(clients)[i].numéro = EFFACÉ
    ) => ancien(clients)[i] = clients[i]
    # Vieillit tous les autres.
    (∀i, MIN_CLIENTS ≤ i ≤ MAX_CLIENTS)
    (
        ancien(clients)[i].numéro ≠ VIDE
      ou
        ancien(clients)[i].numéro ≠ EFFACÉ
    ) => clients[i].p.âge = ancien(clients)[i].p.âge + 1
    # le reste du tableau n'est pas modifié.
    Seul l'âge est modifié.
fin vieillir
```

Cet algorithme n'est pas « élégant ». On y trouve beaucoup de calculs de i + 2 qui le rendent difficile à comprendre. C'est le signe que l'hypothèse de départ est mal choisie. Ces calculs inutiles sont éliminés en modifiant **(H)**. Un second défaut, plus grave, est que cette version ne dit rien de l'invariant ou du variant de contrôle. Vérifier cet algorithme devient alors difficile dans ces conditions.

La modification de l'hypothèse permet de corriger cette situation.

Hypothèse (H') : les âges ont été ajustés dans toutes les cases, jusqu'à celle de numéro i - 1 inclus.

On trouve alors un algorithme différent du précédent, mais bien plus facile à comprendre et à maintenir. L'analyse détaillée de cette nouvelle version est laissée en exercice. On trouve l'algorithme ci-dessous dans lequel n'a pas été répété le début, identique au début de la version précédente.

Algorithme 11 : Vieillir tous les clients de 1 an - Version 2.0

```
Algorithme vieillir
...
initialisation
    i <- MIN_CLIENTS
jusqu'à
    i > MAX_CLIENTS ou sinon clients[i].numéro = VIDE
invariant
    i > MIN_CLIENTS => estVieilli
                (ancien(clients), clients, MIN_CLIENTS, i - 1)
variant de contrôle
    MAX_CLIENTS - i + 1
répéter
    si
        clients[i].numéro ≠ EFFACÉ
    alors
        # Vieillir ce client.
        clients[i].p.âge <- clients[i].p.âge + 1
    sinon
        # Ce client est effacé : ne rien faire.
        rien
    fin si
    i <- i + 1
fin répéter
postcondition
...
```

© ENI Editions - All rigths reserved

Il reste à écrire le prédicat **estVieilli** et à l'utiliser pour exprimer la postcondition d'une façon plus algorithmique. Ce travail complémentaire est laissé en exercice.

On peut continuer à résoudre des exercices dans le même contexte. En voici quelques exemples.

f. Exercices

Exercice 2 : Trouver les clients habitant 'LE MANS'

On veut établir la liste des clients domiciliés dans la ville du MANS pour leur envoyer un prospectus.

1. Écrire l'algorithme qui détermine tous les clients qui habitent 'LE MANS'.

Une promotion est programmée pour les clients qui habitent dans le 75 et âgés entre 40 à 50 ans.

2. Écrire l'algorithme qui sélectionne ces clients.

© ENI Editions - All rigths reserved

Itérer dans un tableau

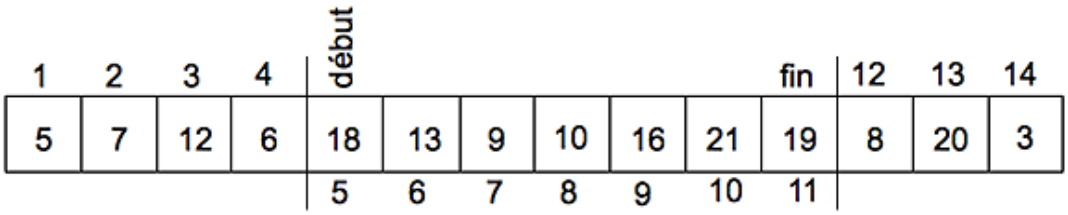
Cette section présente d'autres exemples d'itérations. Ce que nous savons déjà permet d'écrire des solutions complètes. La première partie étudie en détail le problème de la détermination du rang de la composante minimum d'un vecteur. On illustre ainsi, une fois de plus, la recherche linéaire dans un tableau. La deuxième partie développe l'analyse d'un algorithme efficace de recherche dans un tableau trié. Il ne s'agit plus d'une recherche linéaire et l'analyse est, cette fois, moins immédiate.

1. Rang de la composante minimum d'un tableau

On donne un tableau t dont les composantes, c'est-à-dire les contenus des cases, sont d'un type **T** quelconque mais **COMPARABLE**. L'une de ces composantes, au moins, a une valeur inférieure ou égale à la valeur de toutes les autres composantes : c'est la composante minimum. On s'intéresse à son numéro de composante, son rang, dans le tableau.

Faire un algorithme qui calcule le rang de la composante minimum d'un tableau.

Ainsi, pour l'exemple illustré par le dessin de la figure ci-dessous, la fonction retourne 7 puisque table[7] = 9 est la plus petite valeur de table[5] à table[11].



Les spécifications de la fonction sont données ci-dessous.

*Algorithme 12 : Spécifications de la fonction **rangDuMin***

```
Algorithme rangDuMin
    # Le numéro de case de la composante minimum de
    # table[début..fin].
Entrée
    table : TABLEAU[T → COMPARABLE] # La table cible.
    début, fin : ENTIER # L'intervalle de recherche.
Résultat : ENTIER
précondition
    # Le tableau table[début .. fin] a été initialisé.
    (∀i, début ≤ i ≤ fin)(table[i] ≠ NUL)
    # début et fin sont des index valides sur table.
    index_valide(table, début)
    index_valide(table, fin)
    début ≤ fin
postcondition
    min(table, début, fin) = table[Résultat]
fin rangDuMin
```

La postcondition exprime que la composante de numéro **Résultat** est le minimum de `table[début .. fin]` en utilisant l'algorithme **min** qu'il s'agit à présent de spécifier. C'est fait par l'algorithme suivant.

*Algorithme 13 : Spécifications de la fonction **min***

```
Algorithme min
    # La composante minimum de table[début .. fin].
Entrée
    table : TABLEAU[T → COMPARABLE] # La table cible.
    début, fin : ENTIER # L'intervalle de recherche.
Résultat : ENTIER
précondition
```

© ENI Editions - All rigths reserved

```
    # Le tableau table[début .. fin] a été initialisé.
    (∀i, début ≤ i ≤ fin)(table[i] ≠ NUL)
    # début et fin sont des index valides sur table.
    index_valide(table, début)
    index_valide(table, fin)
    début ≤ fin
postcondition
    début = fin => Résultat = table[début]
    début < fin => Résultat =
        inf(table[début], min(table, début+1, fin))
fin min
```

Ainsi, la fonction **min** rend cette fois la valeur de la composante minimum et non plus le numéro de la case qu'elle occupe. Encore une fois, la deuxième clause de la postcondition de cette fonction utilise un nouvel algorithme **inf**. En paraphrasant cette clause, on a que, lorsque `début < fin`, la composante minimum est la plus petite des deux valeurs entre la première composante et la composante minimum du tableau privé de sa première case. Il reste donc encore à spécifier **inf**.

*Algorithme 14 : Spécifications de la fonction **inf***

```
Algorithme inf
    # La valeur du plus petit des deux paramètres.
Entrée
    a,b : T → COMPARABLE # Les valeurs à comparer.
Résultat : T
précondition
    aucune
postcondition
    a ≤ b => Résultat = a
    a > b => Résultat = b
fin inf
```

Étudions la réalisation de **rangDuMin**.

Supposons que nous sachions déterminer le plus petit élément d'un tableau de `i` éléments. *Peut-on trouver le plus petit d'un tableau de i + 1 éléments ?*

- si le $(i + 1)^{ème}$ élément est inférieur au minimum des i premiers éléments, c'est lui qui devient le nouveau minimum ;
- sinon, le minimum déjà trouvé reste le minimum des i + 1 éléments.

Nous savons donc trouver le minimum de i + 1 éléments quand nous connaissons celui de i éléments. Posons donc :

assertion (A) : *nous savons trouver le minimum de i + 1 éléments quand nous connaissons celui de i éléments.*

Or, nous savons trouver le minimum d'un tableau qui n'a qu'une seule composante. Dans le cas d'une composante unique, c'est elle la composante minimum. Voyons pourquoi ceci résout le problème :

- connaissant le minimum pour i = 1 élément, nous savons déterminer le minimum pour i + 1 = 2 éléments d'après l'assertion (A) ;
- connaissant le minimum pour i = 2 éléments, nous savons déterminer le minimum pour i + 1 = 3 éléments d'après (A) ;
- connaissant le minimum pour i = 3 éléments, nous savons déterminer le minimum pour i + 1 = 4 éléments d'après (A) ;
- ...
- connaissant le minimum pour i = n - 1 éléments, nous savons déterminer le minimum pour i = n éléments d'après l'assertion (A).

Faire une hypothèse sur l'état actuel

© ENI Editions - All rigths reserved

Hypothèse (H) : Résultat est le rang du minimum des composantes de table depuis celle de numéro début jusqu'à celle de numéro i - 1 inclus.

On obtient donc :

```
...
table[Résultat] = min(table, début, i - 1)
```

Dans quelles circonstances le problème est-il alors résolu ?

Voir si c'est fini

`i - 1` est le numéro de la dernière composante observée. C'est fini s'il n'existe plus de composante à observer, donc si `i - 1 = fin`, c'est-à-dire si `i = fin + 1`. Comme `i` augmente, c'est fini dès que `i ≥ fin + 1` et donc dès que `i > fin`.

Se rapprocher de l'état final

Dans le cas contraire, il faut observer la case suivante pour avancer vers la solution. La prochaine composante à observer est celle de numéro `i` :

```
...
si
    table[i] < table[Résultat]
alors
    Résultat <- i  # Nouveau minimum.
...
```

Dans tous les cas, la situation a changé : une composante de plus a été observée.

```
# Résultat est le rang du minimum des composantes de table depuis
# celle de numéro début jusqu'à celle de numéro i inclus.
```

Pour retrouver la situation décrite par l'hypothèse initiale, il suffit d'avancer i :

```
...
i <- i + 1
# Résultat est le rang du minimum des composantes de table depuis
# celle de numéro début jusqu'à celle de numéro i - 1 inclus.
```

Nous retrouvons ainsi un état dans lequel l'hypothèse initiale est vraie et il est alors possible d'itérer les mêmes traitements. L'itération se termine car `i` est un entier qui augmente à chaque itération depuis début. Par conséquent, `fin – i + 1` diminue pour atteindre 0 lorsque `i – 1 = fin`. Le variant de contrôle de l'itération est donc `fin – i + 1`.

Initialiser le calcul

Il reste à initialiser le calcul. Il s'agit de réaliser un état dans lequel l'hypothèse est vérifiée. Comme `i – 1` repère la dernière position observée dans la table, c'est que `i` repère la prochaine position à observer. Initialement, la prochaine position à observer est celle de numéro début :

```
...
initialisation
    i <- début
...
```

Il faut aussi donner une valeur initiale à **Résultat** conformément à l'hypothèse de départ. **Résultat** est le numéro de la case qui contient la dernière valeur minimum observée. Dans un tableau d'une seule case, cette valeur est celle de la première case. Ainsi, on peut poser :

```
...
initialisation
    i <- début
    Résultat <- début
...
```

Mais alors, on a `i – 1 = début` puisque c'est `i – 1` qui repère la dernière case observée et donc `i = début + 1`. D'où

© ENI Editions - All rigths reserved

les initialisations définitives :

```
...
initialisation
    i <- début + 1
    Résultat <- début
...
```

Rédiger l'algorithme définitif

Nous avons vu que les traitements doivent être répétés jusqu'à `i > fin`. Nous pouvons aussi remarquer que ce n'est pas fini tant que `i ≤ fin` et donc répéter les traitements **tant que**`i ≤ fin`. L'algorithme est donc :

Algorithme 15 : Recherche du rang de la composante minimum d'un tableau

```
Algorithme rangDuMin
    # Le numéro de case de la composante minimum de table.
Entrée
    table : TABLEAU[T → COMPARABLE] # La table cible.
    début, fin : ENTIER # L'intervalle de recherche.
Résultat : ENTIER
précondition
    # Le tableau table[début .. fin] a été initialisé.
    (∀i, début ≤ i ≤ fin)(table[i] ≠ NUL)
    # début et fin sont des index valides sur table.
    index_valide(table, début)
    index_valide(table, fin)
    début ≤ fin
variable
    i : ENTIER # Numéro de la prochaine composante à observer.
initialisation
    i <- début + 1
    Résultat <- début
jusqu'à
    i > fin
    invariant
        i > début => min(table, début, i - 1) = table[Résultat]
    variant de contrôle
        fin - i + 1
répéter
    si
        table[i] < table[Résultat]
    alors
        Résultat <- i
    fin si
    i <- i + 1
fin répéter
postcondition
    min(table, début, fin) = table[Résultat]
fin rangDuMin
```

Cet algorithme termine l'étude de la recherche du rang de la composante minimum. La recherche de la valeur de la composante est le même problème, à quelques ajustements près. La réalisation de l'algorithme de la fonction **inf** ne fait pas intervenir l'itération. De même, le problème dual, qui s'intéresse à la composante maximum du tableau, se résout de la même façon. Tout ce qui est nécessaire à la résolution de l'exercice suivant est donc déjà en place.

Exercice 3 : Composantes MIN et MAX d'un tableau

Complétons d'abord la solution du problème précédent.

*1. Écrire la solution complète de l'algorithme de la fonction **min**.*

*2. Donner de même une solution complète pour la fonction **inf**.*

On donne un tableau où plusieurs occurrences de la composante minimum apparaissent dans différentes cases.

© ENI Editions - All rigths reserved

*3. Quel est le rang renvoyé par la fonction **rangDuMin** ?*

Les problèmes duaux des précédents se résolvent de la même façon.

*4. Écrire de même les algorithmes complets pour les fonctions **rangDuMax**, **max** et **sup**.*

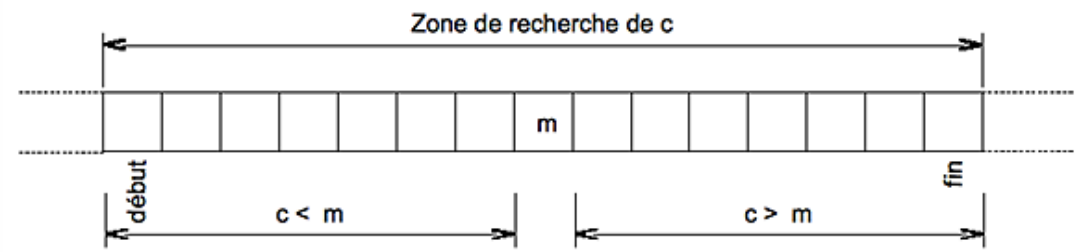
2. Rechercher dans un tableau trié

Les recherches précédentes ne supposaient rien sur les relations mutuelles entre les données enregistrées dans le tableau. Il était parcouru séquentiellement, case par case, depuis la première jusqu'au résultat. Considérons, par exemple, le problème de la recherche du rang d'une composante donnée. Pour un tableau de n composantes, il faut, en moyenne, n/2 accès au tableau pour déterminer le résultat si on ne dispose d'aucun renseignement sur la répartition des données. Cependant, il est possible de répartir les données en tenant compte de l'utilisation du tableau et de la nature des recherches. Ainsi, par exemple, dans un tableau de clients, il est possible de regrouper en tête du tableau les composantes les plus souvent accédées, c'est-à-dire les renseignements qui concernent les clients « les plus fidèles » par exemple. De même, si le tableau regroupe les entrées d'un dictionnaire électronique, on ne placera pas en tête des mots comme 'wagon' ou 'endomorphisme', à moins que le dictionnaire n'ait une destination particulière, comme un dictionnaire destiné à apporter une aide à certains jeux qui valorisent les mots peu utilisés. Cependant, ces considérations concernent l'organisation des données dans le tableau et pas les traitements algorithmiques qui restent les mêmes.

Lorsque les composantes sont triées, en ordre croissant par exemple, le problème peut se résoudre autrement, d'une façon plus efficace, en ce sens que la recherche d'une composante donnée demandera moins d'accès aux éléments du tableau. Bien entendu, la réduction du nombre d'accès nécessite l'étude d'un algorithme adapté : la recherche séquentielle sur un tableau trié reste la même puisqu'elle ne suppose rien sur la répartition des données. La section suivante étudie un algorithme de recherche dans un tableau trié.

a. Recherche par dichotomie

On considère un tableau t dont les composantes sont d'un type **T** quelconque mais **COMPARABLEs**. Les composantes de t sont triées, par exemple en ordre croissant, au moins dans la zone de recherche d'une composante c donnée ou de son rang dans t. Pour rechercher c, on procède de la façon suivante. On regarde d'abord si c se situe dans la case au milieu du tableau. C'est fini avec succès si elle y est. Sinon, c est comparée à la composante m dans la case du milieu. Comme le tableau est trié en ordre croissant, c ne peut se trouver que dans le demi-tableau supérieur si elle est supérieure à m ou dans le demi-tableau inférieur dans le cas contraire. Il est alors possible d'itérer les traitements dans le demi-tableau sélectionné. La figure suivante illustre le procédé.



Cette stratégie de recherche est appelée « recherche par dichotomie » (*binary search*). À chaque étape du calcul, c'est-à-dire pour passer d'un état intermédiaire au suivant, l'espace de recherche est réduit de moitié. C'est un gain considérable par rapport à la recherche linéaire. Alors que pour celle-ci, doubler la taille du tableau double le nombre moyen d'accès au tableau, pour celle-là, doubler la taille du tableau ne fait qu'augmenter d'une unité le nombre d'accès.

Soit **dichotomie** une fonction qui détermine le rang d'une composante quelconque donnée dans un tableau trié en ordre croissant. Ses spécifications sont celles de l'algorithme ci-dessous.

*Algorithme 16 : Spécifications de la fonction **dichotomie** - Version 1.0*

```
Algorithme dichotomie
    # Rang de c dans t[début .. fin] ou ABSENT.
Entrée
    t : TABLEAU[T -> COMPARABLE] # Cible de la recherche.
    début, fin : ENTIER          # Intervalle de recherche.
    c : T                        # Composante à chercher.
Résultat : ENTIER # Numéro de case contenant c ou ABSENT.
précondition
    index_valide(t, début)
    index_valide(t, fin)
```

© ENI Editions - All rigths reserved

```
    début ≤ fin
    # t[début .. fin] est initialisé.
    (∀k, début ≤ k ≤ fin) (t[k] ≠ NUL)
    # t[début .. fin] trié en ordre croissant.
    estTriéAsc(t, début, fin)
postcondition
      (Résultat = ABSENT et (∀k, début ≤ k ≤ fin) (t[k] ≠ c))
    ou sinon
      (début ≤ Résultat ≤ fin et t[Résultat] = c)
fin dichotomie
```

La précondition utilise le prédicat **estTriéAsc** qui rend VRAI si et seulement si le tableau en premier paramètre est trié en ordre croissant dans la zone précisée par les paramètres `début` et `fin`. Les spécifications de ce prédicat sont données plus bas.

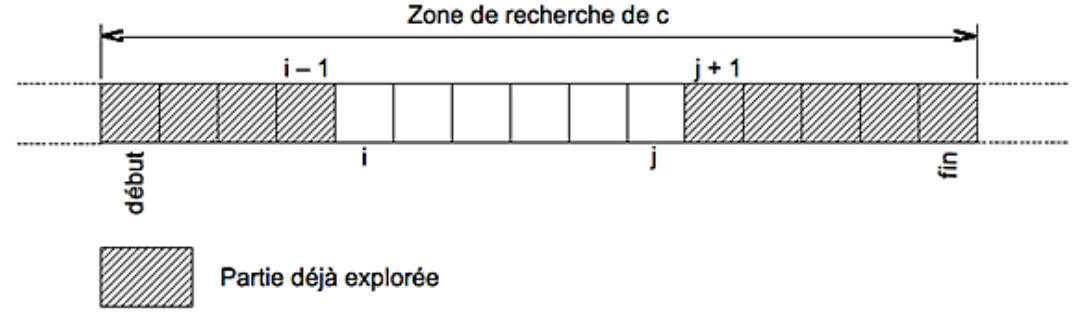
La postcondition est de lecture immédiate. Il est possible d'en donner une expression purement algorithmique, mais qui sera beaucoup plus difficile à lire et donc, à comprendre.

Les spécifications du prédicat **estTriéAsc** sont données par l'algorithme suivant.

*Algorithme 17 : Spécifications de la fonction **estTriéAsc***

```
Algorithme estTriéAsc
    # t[début .. fin] est-il trié en ordre croissant ?
Entrée
    t : TABLEAU[T -> COMPARABLE]
    début, fin : ENTIER
Résultat : BOOLÉEN
précondition
    index_valide(t, début)
    index_valide(t, fin)
    début ≤ fin
    # t[début .. fin] est initialisé.
    (∀k, début ≤ k ≤ fin) (t[k] ≠ NUL)
postcondition
début = fin => Résultat = VRAI
début < fin => Résultat =
    (
        t[début] ≤ t[début+1]
      et alors
        estTriéAsc(t, début+1, fin)
    )
fin estTriéAsc
```

Il reste à écrire la réalisation de **dichotomie**. Commençons par observer la composante au milieu du tableau. Son numéro milieu est obtenu en calculant le quotient entier de (`début + fin`) dans la division par 2. Ce numéro est le résultat attendu si la case de numéro milieu contient `c`. Sinon, il faut placer `c` par rapport au contenu de la case au milieu du tableau et sélectionner ainsi le demi-tableau dans lequel va se poursuivre la recherche. À mesure que cette recherche avance et à chaque étape, la longueur du tableau à explorer est réduite de moitié. Dans un état intermédiaire, la situation est représentée par la figure suivante :



Cette figure montre un état dans lequel les parties `t[début .. i – 1]` et `t[j + 1 .. fin]` ont été éliminées de la recherche. La partie `t[i .. j]` reste à explorer. Nous pouvons donc préciser l'hypothèse de travail. Commençons par une version incorrecte.

© ENI Editions - All rigths reserved

Faire une hypothèse sur l'état actuel

Hypothèse (H) : c n'est ni dans t[début .. i - 1], ni dans t[j + 1 .. fin]. Résultat = ABSENT.

Voyez-vous pourquoi cette hypothèse n'est pas satisfaisante ? Exploitons-la pour voir où elle nous mène.

Voir si c'est fini

C'est fini lorsque les zones déjà explorées se rejoignent, donc lorsque `i - 1 = j` :

```
...
C'est fini et c n'est pas trouvé lorsque i - 1 =j
```

On peut aussi caractériser cette situation par `i = j + 1`. Lorsque ce n'est pas fini, c'est qu'il reste encore des cases à explorer.

Se rapprocher de l'état final

Le tableau doit encore être exploré de la case de numéro `i` jusqu'à la case de numéro `j`. La stratégie adoptée consiste à diviser le tableau en deux demi-tableaux. Pour cela, il suffit de calculer le numéro de la case centrale, puis de situer la valeur cherchée dans les deux demi-tableaux ainsi définis :

```
...
# Numéro de la case du milieu.
milieu <- quotient(i + j, 2)
# milieu est-il le numéro de case cherché ?
si
    t[milieu] = c
alors
    c'est fini : trouvé en milieu
sinon
    # Placer c par rapport à t[milieu]
    si
        t[milieu] < c
    alors
        c est dans le demi-tableau droit
    sinon
        c est dans le demi-tableau gauche
    fin si
fin si
...
```

Pour exprimer que `c est dans le demi-tableau droit`, il suffit de faire en sorte que le demi-tableau inférieur soit éliminé de la recherche. C'est `i - 1` qui donne la limite du demi-tableau gauche. On retrouve donc **(H)** en ajustant `i - 1` pour qu'il soit égal à `milieu`. On aura `i - 1 = milieu` lorsque `i = milieu + 1` :

```
...
# c est dans le demi-tableau droit.
i <- milieu + 1
...
```

De même, `c est dans le demi-tableau gauche` s'exprime en écrivant :

```
...
# c est dans le demi-tableau gauche.
j <- milieu - 1
...
```

Initialiser le calcul

(H) précise que toutes les cases jusqu'à celle de numéro `i - 1` et depuis celle de numéro `j + 1` ont déjà été observées. La prochaine case à observer dans le demi-tableau gauche est celle de numéro `i`. Initialement, c'est `début` :

```
...
```

© ENI Editions - All rigths reserved

```
initialisation
    i <- début
...
```

De même, la prochaine case à observer dans le demi-tableau droit est celle de numéro `j`. Initialement c'est fin :

```
...
initialisation
    j <- fin
...
```

Enfin, on cherche encore et donc, la case contenant `c` n'a pas encore été trouvée :

```
...
initialisation
    Résultat <- ABSENT
...
```

Les initialisations complètes sont donc les suivantes :

```
...
initialisation
    i <- début
    j <- fin
    Résultat <- ABSENT
...
```

Rédiger l'algorithme définitif

Il reste à exprimer la condition d'arrêt de la recherche, l'invariant et le variant de contrôle de l'itération.

C'est fini et `c` n'a pas été trouvé lorsque `i - 1 = j`, donc lorsque `i = j + 1`, soit dès que `i ≥ j + 1` ou encore dès que `i > j`. Ainsi, les traitements sont à itérer jusqu'à ce que `i > j` ou alors lorsque `c` a été trouvé, c'est-à-dire lorsque le résultat n'est plus ABSENT. Dans un état intermédiaire quelconque, l'hypothèse **(H)** à maintenir s'écrit :

```
invariant
    c < t[i] ou t[j] < c => Résultat = ABSENT
```

La recherche est donc en échec dans les deux demi-tableaux lorsque `i` et `j` sont entre `début` et `fin` :

```
invariant
    i > début et j < fin =>
        (
            non estDans(t, début, i - 1, c)
          et
            non estDans(t, j + 1, fin, c)
        )
...
```

Enfin, `i` augmente et `j` diminue jusqu'à `i - 1 = j` d'après la condition d'arrêt. Ainsi, `j - i + 1` diminue vers 0. C'est le variant de contrôle. Il mesure le nombre de cases qui n'ont pas encore été observées dans la région `t[i .. j]`. L'algorithme ci-dessous est une version provisoire dans laquelle on ne répète pas la précondition et la postcondition.

*Algorithme 18 : Définition de la fonction **dichotomie** - Version provisoire*

```
Algorithme dichotomie
    # Le rang de c dans t[début .. fin].
Entrée
    t : TABLEAU[T → COMPARABLE] # La cible de la recherche.
    début, fin : ENTIER # Numéros des cases du domaine de recherche.
    c : T               # La composante cherchée.
Résultat : ENTIER # Le numéro de la case de t[début .. fin ]
                  # qui contient c ou sinon ABSENT.
précondition
...
variable
```


© ENI Editions - All rigths reserved

```
    i, j   : ENTIER # Numéros des cases entre lesquelles chercher.
    milieu : ENTIER # Numéro case milieu du domaine de recherche.
initialisation
    i <- début
    j <- fin
    Résultat <- ABSENT
invariant
    i > début et j < fin =>
        (
            non estDans(t, début, i - 1, c)
          et
            non estDans(t, j + 1, fin, c)
        )
variant de contrôle
    j - i + 1
jusqu'à
    i > j ou Résultat ≠ ABSENT
répéter
    # Numéro de la case au milieu du domaine de recherche.
    milieu <- quotient(i + j, 2)
    # milieu est-il le numéro de la case cherchée ?
    si
        t[milieu] = c
    alors
        # C'est fini : c trouvé dans la case de numéro milieu.
        Résultat <- milieu
    sinon
        # Situer c par rapport à t[milieu].
        si
            t[milieu] < c
        alors
            # c n'est pas dans le demi-tableau gauche.
            i <- milieu + 1
        sinon
            # c n'est pas dans le demi-tableau droit.
            j <- milieu - 1
        fin si
    fin si
fin répéter
postcondition
...
fin dichotomie
```

Mais que se passe-t-il lorsque `c` n'est pas une composante du domaine de recherche, autrement dit lorsque `c` n'est pas une composante de `t[début .. fin]` ? L'algorithme précédent parcourt malgré tout le tableau jusqu'à ce que `i > j` et se termine avec **Résultat** = ABSENT. Cependant, lorsque `c < t[début]` ou que `c > t[fin]`, l'exploration de `t[début .. fin]` est inutile puisque `t` est trié en ordre croissant par hypothèse. On a, en effet :

`(c < t[début] ou c > t[fin]) => ((∀k, début ≤ k ≤ fin) (c ≠ t[k]))`

Appelons (1) cette équation. Elle exprime que c n'est pas dans t lorsque `c < t[début]` **ou** `c > t[fin]`. Dans ce cas, le parcours de `t[début .. fin]` est inutile, bien que le résultat obtenu soit correct. Doit-on se préoccuper d'obtenir une solution mieux construite alors que celle-ci donne le bon résultat ? Sans doute oui, car ce qui ne va pas dans cet algorithme, c'est qu'il est construit sur une hypothèse **(H)** qui ne décrit pas correctement l'état intermédiaire. Comme tout le reste s'en déduit, on ne peut pas faire confiance à cette solution.

(H) énonce que c n'est pas encore trouvé, mais si on cherche encore, c'est que nous ne savons pas si c est ou non une composante de t. Or, lorsque `c < t[début]` ou que `c > t[fin]`, on sait que c n'est pas dans `t[début .. fin]` d'après l'équation (1). Reformulons donc **(H)** : il est inutile de chercher si la précondition exprimée par (1) est satisfaite. Par conséquent, si la recherche n'est pas terminée, c'est que cette équation exprime une condition qui n'est pas satisfaite :

Hypothèse (H') : t[début] ≤ c ≤ t[fin] et c n'est ni dans t[début .. i - 1], ni dans t[j + 1 .. fin]. Résultat = ABSENT.

Ce qui change, ce sont les initialisations et l'invariant. Pour réaliser l'état initial, il faut d'abord s'assurer que `t[début] ≤ c ≤ t[fin]` :

© ENI Editions - All rigths reserved

```
...
Résultat <- ABSENT # Encore rien trouvé.
si
    t[début] ≤ c ≤ t[fin]
alors
    # c ne satisfait pas la précondition de l'équation (1).
    ... comme l'algorithme précédent.
sinon
    # c ∉ t[début .. fin].
    rien
fin si
...
```

L'invariant exprime que t[début] ≤ c ≤ t[fin] quand on cherche dans t :

```
...
invariant
    t[début] ≤ c ≤ t[fin]
  et ... comme l'algorithme précédent.
...
```

L'algorithme définitif est donc le suivant, dans lequel ne sont pas répétées la signature et les déclarations :

*Algorithme 19 : Définition de la fonction **dichotomie** - Version définitive*

```
...
initialisation
    Résultat <- ABSENT
si
    t[début] ≤ c ≤ t[fin]
alors
    i <- début
    j <- fin
    invariant
        t[début] ≤ c ≤ t[fin]
      et
        (
            i > début et j < fin =>
                (
                    non estDans(t, début, i - 1, c)
                  et
                    non estDans(t, j + 1, fin, c)
                )
        )
    jusqu'à
        i > j ou Résultat ≠ ABSENT
        variant de contrôle
            j - i + 1
    répéter
        # Numéro de la case au milieu du domaine de recherche.
        milieu <- quotient(i + j, 2)
        # milieu est-il le numéro de la case cherchée ?
        si
            t[milieu] = c
        alors
            # C'est fini : c trouvé dans la case de numéro milieu.
            Résultat <- milieu
        sinon
            # Situer c par rapport à t[milieu].
            si
                t[milieu] < c
            alors
                # c n'est pas dans le demi-tableau gauche.
                i <- milieu + 1
            sinon
```


© ENI Editions - All rigths reserved

```
                # c n'est pas dans le demi-tableau droit.
                j <- milieu - 1
            fin si
        fin si
    fin répéter
fin si
...
```

Le problème algorithmique est résolu. L'informaticien programmeur cherchera ensuite à implémenter cette solution dans un langage particulier, pour une architecture matérielle particulière. C'est un autre problème, qui n'entre pas dans le cadre des préoccupations de ce livre. Cependant, nous pouvons faire quelques remarques au sujet de l'implémentation de cet algorithme.

Le calcul de milieu est exprimé ici par :

```
milieu <- quotient(i + j, 2)
```

qui indique que milieu est le quotient entier dans la division euclidienne de i + j par 2. Il est clair que l'implémentation de ce calcul n'utilisera probablement pas une fonction **quotient** mais les opérations arithmétiques correspondantes. Ainsi, en langage C par exemple, on écrirait :

```
int i, j, milieu ;
...
milieu = (i + j) / 2 ;
...
```

La première instruction déclare trois entiers. L'instruction suivante affecte à milieu la demi-somme des entiers i et j. Cependant, tout entier est représenté en machine par une donnée dont le domaine est restreint à un sous-ensemble des entiers. Cette restriction est imposée par la technologie des calculateurs. Ainsi, l'addition i + j peut conduire à un résultat qui n'est plus dans le domaine des entiers représentables en machine. C'est le cas, par exemple, dès que i et j sont deux nombres strictement supérieurs à la moitié de la borne supérieure des entiers représentables. Il en résulte un dépassement de capacité et, au mieux, un diagnostic d'erreur du processeur du langage utilisé, au pire un calcul qui continue avec un résultat faux. Ce type de problème se présente fréquemment en implémentation , avec d'autres difficultés plus subtiles et incomparablement plus difficiles à résoudre. Cet exemple montre que, l'algorithme étant au point, le problème informatique n'est pas pour autant résolu.

Revenons à notre exemple. Il existe différentes solutions pour éviter les dépassements de capacité dans ce cas. L'une d'elles consiste à calculer la valeur de milieu en ne réalisant que des opérations sûres, c'est-à-dire dont on vérifie qu'elles ne provoqueront pas de dépassement. Ainsi, par exemple, il est facile de vérifier que :

```
quotient(i + j, 2) = i + quotient(j - i, 2)
```

On programmera donc :

```
milieu = i + (j - i) / 2 ;
```

ce qui garantit que les opérations ne conduisent à aucun dépassement de capacité des entiers résultats. La soustraction ne provoquera pas de débordement puisque i et j sont deux entiers positifs ou nuls et que, de plus, i ≤ j dans le domaine de calcul de milieu. Une autre solution consiste à exploiter les opérateurs binaires fournis pas certains langages, dont le langage C. Une division par 2 consiste à réaliser un décalage d'un rang vers la droite des chiffres binaires du nombre à diviser. C et les langages dérivés fournissent pour cela l'opérateur >> :

```
milieu = (i + j) >> 1 ;
```

Dans ce cas, les propriétés de l'opération de décalage à droite font que le dépassement de capacité sur la somme, si elle ne provoque pas d'erreur, n'a pas de conséquence néfaste sur le résultat du calcul de milieu.

b. Extensions

Exercice 4 : Compléments

*1. Écrire une réalisation itérative de **estTriéAsc**. Modifier l'algorithme pour l'adapter à un tableau trié en ordre décroissant.*

*2. Écrire une réalisation récursive de la fonction **dichotomie**.*

© ENI Editions - All rigths reserved

Algorithme ou programme ?

Nous en savons suffisamment à présent pour revenir sur quelques notions qui ont déjà été abordées dans les chapitres Qu'est-ce que l'algorithmique ? et Programmes directs. Qu'est-ce qu'un algorithme ? Qu'est-ce qui le distingue d'un programme destiné à un ordinateur ? Construire un programme et définir un algorithme sont-elles des activités distinctes, chacune développant ses propres méthodes, ou ne constituent-elles que les « deux faces de la même médaille » ? Cette section ne cherche pas à répondre de façon définitive. On y expose seulement quelques idées simples qui complètent la vision partielle des chapitres précédents. La première partie développe un exemple recopié d'un livre qui traite d'algorithmique. La seconde partie expose l'une des solutions préconisées pour le problème.

1. Un exemple édifiant

On a initialisé un tableau t de 100 composantes réelles positives avec n nombres, 0 ≤ n ≤ 100 dont on veut déterminer la moyenne arithmétique. Dans un livre que je n'aurai pas la cruauté de citer, je lis la solution suivante :

```
VAR
    marques  : TABLEAU[1 .. 100] de RÉEL ;
    nombres  : ENTIER ;
    somme, i : ENTIER ;
    moyenne  : RÉEL   ;
DÉBUT
    (* Remplir le tableau *)
    i := 1 ;
    écrire("Donner un entier : ") ;
    lire(marque[i]) ;
    TANT QUE(marques[i] < > - 1)
      DÉBUT
        i := i + 1 ;
        écrire("Quel est l'entier suivant ? ") ;
        lire(marques[i]) ;
      FIN ;
    (* Calculer la moyenne *)
    nombres := i ;
    i := 1 ;
    somme := 0 ;
    TANT QUE(i < > nombres)
      DÉBUT
        somme := somme + marques[i] ;
        i := i + 1 ;
      FIN ;
    moyenne := somme / nombres ;
    (* Afficher le résultat *)
    écrire("La moyenne est ", moyenne) ;
FIN.
```

Les constructions *(* … *)* indiquent un commentaire. L'opérateur := est l'opérateur d'affectation qui initialise la variable à gauche par le résultat de l'évaluation de l'expression à droite. Les autres constructions syntaxiques ne devraient pas poser de problème de compréhension de ce texte. La présentation du texte d'origine a été légèrement modifiée pour en améliorer la lecture, mais ces modifications n'affectent pas les conclusions de la discussion qui suit. Bien entendu, vous avez déjà repéré les erreurs essentielles majeures de cette solution. Cette solution ressemble à une caricature, et pourtant…

On peut faire, à cette façon d'envisager le problème, deux familles de reproches qui ne sont pas de même nature, ni surtout de même importance : *les maladresses et les fautes*, celles-ci expliquant en partie celles-là.

a. Quelques maladresses : les notations

Passons rapidement sur quelques maladresses vénielles, comme par exemple les formes syntaxiques qui semblent obligatoires : le point-virgule systématique en fin de ligne, l'opérateur '< >' au lieu de '≠', **DÉBUT .. FIN** pour délimiter les blocs.

De même que la recette de cuisine n'est pas la préparation du plat, l'algorithmique se distingue de la programmation en ce que leurs objectifs, leurs exigences, leurs méthodes et leurs outils ne sont pas les mêmes. La première cible une machine abstraite dont le « processeur » serait le cerveau humain. L'autre concerne une machine

© ENI Editions - All rigths reserved

virtuelle infiniment plus rudimentaire. Ainsi, par exemple, un compilateur de langage analyse un texte de programme dans lequel on veut pouvoir utiliser librement l'espace pour le présenter lisiblement. Il exige donc, souvent mais pas toujours, un séparateur, usuellement le point-virgule, pour distinguer les instructions les unes des autres. Pourquoi cette complication dans le texte d'un algorithme lorsqu'il n'y a pas d'ambiguïté possible pour le « processeur » de la machine abstraite auquel il est destiné ? De même, pourquoi utiliser '< >' au lieu du symbole '≠' usuel en Mathématiques ? Mais ce sont des détails. Après tout, l'une des thèses de ce livre affirme que la syntaxe est libre, alors pourquoi pas 'a < > b' au lieu de 'a ≠ b' ? Le clavier d'un ordinateur ne permet pas toujours de présenter les symboles mathématiques, aussi, laissons de côté ces reproches.

Un deuxième type de maladresses est la notation adoptée qui est plus « gênante ». Le texte est rédigé dans un langage qui ressemble beaucoup au langage de programmation PASCAL dont les mots-clé auraient été francisés. Quel peut bien être l'intérêt de cette pratique ? La pauvreté lexicale d'un langage de programmation n'est pas un frein insurmontable à la compréhension de son texte. Ce n'est pas cette distance aux pratiques grammaticales quotidiennes du lecteur qui peut l'empêcher d'appréhender immédiatement le texte de l'algorithme écrit directement dans le langage cible. Pour la compréhension de la solution, la notation ci-dessous est rigoureusement équivalente à la précédente :

```
VAR
    marques  : ARRAY[1 .. 100] OF REAL ;
    nombres  : INTEGER ;
    somme, i : INTEGER ;
    moyenne  : REAL    ;
BEGINx
    (* Remplir le tableau *)
    i := 1 ;
    WRITE("Donner un entier : ") ;
    READ(marque[i]) ;                          (* R1 *)
    WHILE(marques[i] < > - 1) DO
      BEGIN
        i := i + 1 ;
        WRITE("Quel est l'entier suivant ? ") ;
        READ(marques[i]) ;                     (* R2 *)
      END ;
    (* Calculer la moyenne *)
    nombres := i ;
    i := 1 ;
    somme := 0 ;
    WHILE(i < > nombres) DO
      BEGIN
        somme := somme + marques[i] ;
        i := i + 1 ;
      END ;
    moyenne := somme / nombres ;
    (* Afficher le résultat *)
    WRITE("La moyenne est ", moyenne) ;
END.
```

On peut alors faire l'économie de cette complication inutile qui consiste à écrire d'abord un *programme* en Français pour ensuite en traduire les mots-clés en Anglais. Là encore, ce n'est pas l'essentiel. La syntaxe d'un algorithme étant libre, pourquoi pas du « French Pascal » ? Sauf que, cette fois, une telle pratique fait des ravages chez les apprentis programmeurs. Ils ne tardent pas à comprendre qu'on les invite à un jeu stérile et même nuisible en ce qu'il distrait de l'essentiel : *écrire un programme juste* ! On leur demande d'écrire deux fois le même texte et ils ont raison : pourquoi faire d'abord un programme en Français alors qu'il est si simple de l'écrire d'emblée dans le bon langage ? Pour écrire un algorithme, il FAUT utiliser une notation libre, mais claire et concise, qui met bien en évidence le séquencement des actions à réaliser en fonction des assertions qui décrivent les états intermédiaires du système. Nous y reviendrons plus bas.

Une maladresse du même ordre est le mélange, dans le corps de l'algorithme, de clauses destinées à calculer une moyenne et d'instructions d'entrée/sortie : **écrire**, **lire**. Quel est la responsabilité de cet algorithme : calculer une moyenne ? Réaliser les interfaces entre un utilisateur et un programme ? Les instructions d'entrée/sortie, à quelques rares exceptions qui concernent la conception d'interfaces, sont des instructions de programmation : elles n'ont rien à faire dans un algorithme de calcul ! Ici, nous voulons calculer une moyenne arithmétique :

```
moyenne(la structure de données en entrée) : RÉEL
...
    < clauses réalisant le calcul >
...
```

C'est à la fonction **moyenne** d'interroger un tableau, le résultat d'une requête à une base de données… pour effectuer ce calcul. Changer la structure de données en entrée ne modifie, éventuellement, que la définition de cet

© ENI Editions - All rigths reserved

algorithme et la construction de cette structure ne le concerne pas.

L'algorithme, le *programme destiné à la machine abstraite*, exprime *ce que fait* le programme abstrait et, accessoirement, *comment* il le fait. Le programme destiné à la machine concrète ne dit quasiment rien sur ce qu'il fait : il ne dit que *comment* il réalise certaines transformations sur les données. C'est cette particularité de la programmation qui exige une étude préalable soignée de la définition des transformations à faire subir aux données. Or, les valeurs de ces dernières caractérisent l'état dans lequel se trouve le système logiciel et c'est le contrôle de la succession de ces états qui permet de vérifier la correction d'un programme : converge-t-il vers l'état dans lequel les valeurs des variables représentent l'état final attendu ? Encore une fois, le *comment* est accessoire dans un algorithme : c'est de la programmation et les langages inventés dans ce but l'expriment très bien. Il n'y a pas lieu de les modifier pour cela. L'algorithme, lui, a pour responsabilité de mettre en évidence les états successifs du système logiciel, depuis l'état initial I jusqu'à l'état final F, en exprimant les assertions construites sur les valeurs des variables qui réalisent les états successifs.

Reprenons notre algorithme de calcul. Il doit dire ce qu'il fait. La signature exprime d'abord qu'il s'agit d'une *fonction*. C'est la notation : **RÉEL** placée en fin de signature qui l'indique : cet algorithme calcule un résultat qui est un nombre réel. Il ne modifie pas les données qu'il reçoit pour réaliser ce calcul : c'est une propriété implicite, mais ce n'est pas la seule convention possible. De même, au lieu des notations précédentes, il est possible d'écrire, par exemple :

```
moyenne
Entrée
    un tableau marques de nombres réels positifs ou nuls
    un entier n ≥ 0 qui est le nombre de réels dans marques
Résultat
    n > 0 => la moyenne arithmétique réelle des composantes de
    numéros 1 à n de marques
    n = 0 => INFINI
```

Que se passera-t-il si cet algorithme doit effectuer son calcul à partir de données calculées et que les calculs préalables ont donné n < 0 ? Les calculs préalables ne sont pas de sa responsabilité. De même, ce n'est pas à lui de savoir si l'état du système logiciel est un état valide pour faire appel à ses services. C'est un algorithme de calcul d'une moyenne arithmétique, qui sait faire ce calcul pour un nombre de composantes positif ou nul dans un tableau de nombres réels. Il ne peut pas faire moins ; on ne peut pas lui en demander plus. Cependant, ces conditions doivent être précisées. C'est l'objet de la précondition. Elle dit dans quel état doit se trouver le système pour que l'algorithme garantisse les résultats qu'il produit. Le tableau de réels doit exister, avoir été déclaré et initialisé avec des nombres réels positifs ou nuls, ici entre les composantes de numéros 1 à n :

```
...
précondition
    marques ≠ NUL
    n ≥ 0
    index_valide(marques, 1)
    index_valide(marques, n)
    (∀ i, 1 ≤ i ≤ n)(marques[i] ≥ 0)
...
```

Bien entendu, on précise ainsi un état initial. Il est possible de le faire avec d'autres notations :

```
état initial
    n ≥ 0
    marques[1 .. n] est défini (toutes les composantes ont une valeur
    légale) et chaque composante est un réel positif ou nul
...
```

La précondition est un prédicat qui est fait, sauf mention contraire, de la conjonction de clauses dont la valeur est soit VRAI, soit FAUX. Autrement dit, chaque clause de la précondition est reliée aux autres clauses par un **et** logique. La précondition prend la valeur FAUX si l'une au moins des clauses qui la composent prend la valeur FAUX. Alors, l'algorithme s'autorise à faire « ce qu'il veut » : renvoyer un résultat nul, renvoyer un résultat quelconque mais non nul, provoquer l'arrêt brutal des calculs quand il sera implémenté, ne rien faire… On ne sait pas : « c'est lui qui décide ».

Ces conditions étant remplies, il s'agit, à présent, de préciser l'état final. C'est fait dans la postcondition. Elle utilise, comme la précondition et sauf mention contraire, une conjonction de clauses pour décrire l'état final.

```
postcondition
    n > 0 => Résultat = moyenne arithmétique des composantes de
    numéros 1 à n du tableau marques
    n = 0 => Résultat = INFINI
```

Là encore, on peut faire autrement et utiliser, par exemple, le symbolisme mathématique qui est bien plus facile à

© ENI Editions - All rigths reserved

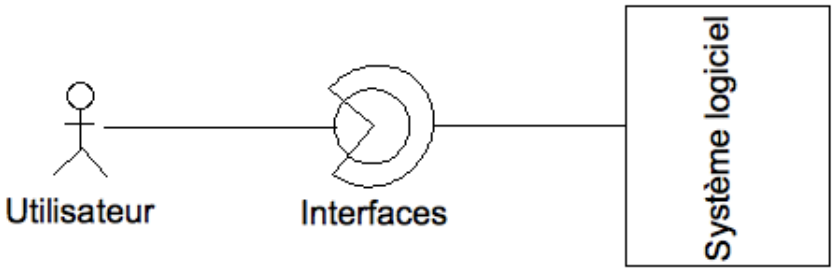
lire et à comprendre d'emblée, avec une culture réduite dans ce domaine.

La signature de l'algorithme et le commentaire qui l'accompagne, la précondition et la postcondition constituent la partie publique du module logiciel. Nous l'avons déjà souligné : c'est sa documentation. Il s'agit des renseignements communiqués à tout utilisateur de ses services. Ces renseignements sont nécessaires et suffisants à un usage averti de l'algorithme. Cependant, le corps de l'algorithme n'est pas nécessaire à sa documentation. On doit pouvoir s'en passer car ce qu'il dit, pour l'essentiel, c'est *comment* il fait pour calculer une moyenne. D'ailleurs, ce contenu n'est pas nécessairement disponible lorsque c'est l'implémentation d'un algorithme qui est utilisée dans un développement. Cette implémentation particulière est souvent un composant commercial dont le code source est protégé et inaccessible. C'est ce qui explique que, parfois, la clause qui exprime une postcondition semble copier une instruction du corps de l'algorithme :

```
...
x <- a + b
...
postcondition
    x = a + b
...
```

C'est que, pour l'utilisateur de l'algorithme, la seule partie visible est le bloc de la postcondition. Il n'est pas nécessaire de lui communiquer comment est obtenu cet état. D'ailleurs, il est possible que, dans une autre version, cet état soit obtenu autrement. Un chapitre ultérieur montrera différentes réalisations d'un algorithme de tri des composantes d'un tableau qui, toutes, seront définies par la même spécification.

Pour résumer ce qui vient d'être développé dans cette section, voyez le dessin suivant.



On y voit représentés trois domaines : un système logiciel qui réalise un calcul au sens donné à ce mot au chapitre Qu'est-ce que l'algorithmique ?, un utilisateur, c'est-à-dire un client, logiciel ou humain, des services du système et des interfaces entre l'utilisateur et le logiciel. Le domaine de l'algorithme, sauf cas particuliers certes importants mais cependant marginaux, est celui du système logiciel pour lequel il doit exprimer clairement les responsabilités. Le reste relève de la programmation.

Pour revenir à cet « algorithme » de calcul de moyenne, tout ce qui précède relève du détail comparé aux fautes qu'il contient : ce programme est faux de la pire façon qui soit : il donne « presque toujours » un résultat plausible. L'utiliser avec un grand nombre de composantes risque de ne jamais révéler les erreurs de programmation. Pourtant, une analyse soignée aurait permis d'éliminer les erreurs les plus grossières. Voyons d'abord pourquoi il est faux. La section suivante exposera comment obtenir une solution correcte.

b. Quand les fautes font oublier les maladresses

Le programme proposé est fait de trois segments. Il remplit un tableau de nombres réels en les comptant ; il explore ce tableau pour calculer la moyenne des nombres qu'il contient ; enfin, il écrit la moyenne calculée, sans doute sur un écran. Pour remplir le tableau, il initialise puis itère un même traitement :

- incrémenter le numéro `i` de la prochaine case à initialiser ;
- écrire un message d'appel ;
- lire une valeur et la ranger dans la case de numéro `i` du tableau.

La condition d'arrêt de l'itération est la valeur rangée dans la case de numéro `i` : la saisie se termine lorsque cette valeur est -1. C'est la faute majeure de ce programme. Par une hypothèse implicite, qui aurait du être formulée clairement, `i` est le numéro de la prochaine case à remplir. En effet, il est initialisé à 1 alors qu'aucune valeur n'a encore été placée dans le tableau. La confusion entre calcul et entrée/sortie réalise, par une même instruction, la lecture d'une donnée et son transfert dans le tableau. C'est fait à la ligne étiquetée *(* R1*)* d'abord, puis à la ligne *(* R2 *)* pour les valeurs successives. Ainsi, on obtient :

© ENI Editions - All rigths reserved

```
...
(* marques[1 .. i - 1] a été initialisé. La prochaine case à
   remplir a le numéro i *)
écrire( ... )
lire(marques[i])
...
```

Mais lorsque l'opérateur termine la saisie, il entre -1 qui est compté comme toute autre valeur et rangé dans le tableau. On s'attend alors à ce que le programme corrige, mais ce n'est pas le cas. Ainsi, la moyenne est ensuite calculée sur un nombre de valeurs trop élevé d'une unité et avec une valeur parasite égale à -1 qui diminue d'autant la somme des valeurs. Ce programme est définitivement faux et pour des raisons qui perdurent dans l'apprentissage et les pratiques de la programmation depuis la préhistoire de l'informatique, et cela, malgré les leçons des maîtres du domaine : on programme au lieu d'étudier le problème. On dit *comment* faire au lieu de dire *quoi* faire. Au passage, on remarque aussi que la confusion des rôles est sans doute dictée par une confusion probablement plus profonde sur les concepts. Les langages de programmation proposent deux familles de modules logiciels : les procédures et les fonctions. Nous avons déjà longuement précisé la distinction fondamentale entre eux au chapitre Programmes directs. La responsabilité de la procédure est de modifier des variables d'état. Celle d'une fonction est de calculer un résultat. Comme une opération mathématique, une fonction ne modifie pas ses opérandes. Que dire du programme proposé ? Le module **lire** récupère une valeur obtenue de l'utilisateur. En ce sens, c'est une fonction : elle interroge le système externe et en obtient une valeur. Cependant, elle reçoit en paramètre une variable qu'elle modifie, qu'elle initialise. En ce sens, c'est une procédure. Que certains langages permettent ou simplement s'accommodent de telles pratiques doit justement nous rendre vigilants : il faut de toute nécessité distinguer les différentes responsabilités jusque dans notre façon d'écrire les modules logiciels. C'est la seule façon de nous assurer que nous contrôlons bien les changements d'états.

Continuons sur les fautes. L'utilisateur peut d'emblée saisir -1 pour la première valeur. Aucune valeur pertinente n'aura alors été saisie et le programme calculera cependant une moyenne : c'est l'erreur mise en évidence précédemment. Si nous modifions ce programme pour éliminer cette valeur parasite, alors `n = 0` et le programme se termine par une division par 0. Bref, il semble bien qu'il n'y ait rien à sauver de cette solution. Quand bien même quelques « rustines » rétabliraient un semblant de programme correct, quelle confiance pourrait-on accorder au résultat ? La section suivante étudie une solution correcte. Commençons par le calcul de la moyenne.

2. Une solution au problème de la moyenne

Le calcul de la moyenne est réalisé par une fonction, déjà partiellement définie précédemment, qui divise la somme des composantes du tableau `marques` par le nombre de composantes additionnées. Il n'en faut pas plus pour écrire cette fonction. C'est fait par l'algorithme ci-dessous.

*Algorithme 20 : Définition de la fonction **moyenne***

```
Algorithme moyenne
    # La moyenne arithmétique de marques[1 .. n].
Entrée
    marques : TABLEAU[ RÉEL]
    n       : ENTIER # Nombre de composantes de marques.
Résultat : RÉEL
précondition
    # Les composantes de marques sont numérotées de 1 à n.
    index_valide(marques, 1)
    index_valide(marques, n)
    # Les composantes de marques[1 .. n] ont été initialisées.
    (∀k, 1 ≤ k ≤ n)(marques[k] ≠ NUL)
réalisation
    si
        n = 0
    alors
        Résultat <- INFINI
    sinon
        Résultat <- somme(marques, 1, n) / n
    fin si
postcondition
    n = 0 => Résultat = INFINI
    n > 0 => Résultat = somme(marques, 1, n) / n
fin moyenne
```

© ENI Editions - All rigths reserved

C'est au programme de définir INFINI et donc de donner un sens au calcul de la moyenne lorsque le nombre de constituants de cette moyenne est nul. Remarquez aussi que la clause qui concerne le domaine des valeurs des composantes du tableau `marques` a disparu. C'est qu'un calcul de moyenne est valide pour des réels quelconques, pas nécessairement positifs ou nuls.

Le calcul fait intervenir une fonction **somme** qui rend la somme des composantes du tableau qu'elle reçoit en paramètre. Pour rester général, elle reçoit, en plus du tableau, les numéros de la première et de la dernière case sur lesquelles elle opère. Il est possible de préciser sa signature et la précondition :

```
Algorithme somme
    # Somme des composantes de t[début .. fin].
Entrée
    t : TABLEAU[RÉEL]    # La cible du calcul de la somme.
    début, fin : ENTIER # Num. composantes extrêmes à additionner.
Résultat : RÉEL
précondition
    # début et fin sont des numéros de cases valides.
    index_valide(t, début)
    index_valide(t, fin)
    début ≤ fin
    # Les composantes de t[début .. fin] ont été initialisées.
    (∀k, début ≤ k ≤ fin)(t[k] ≠ NUL)
```

Étudions à présent comment définir cette fonction.

Il s'agit d'ajouter à la valeur de la première case, celle de numéro `début`, la valeur de la case suivante puis de recommencer. Soit **(H)** l'hypothèse qui décrit un état intermédiaire dans lequel un certain nombre de cases ont déjà été additionnées. Supposons que ce soit le cas jusqu'à la case de numéro `i - 1` incluse, de sorte que la prochaine case à additionner, si elle existe, soit celle de numéro `i`.

Faire une hypothèse sur l'état actuel

Hypothèse (H) : **Résultat** est la somme des cases de marques[début .. i - 1].

Voir si c'est fini

C'est fini lorsque `i - 1 = fin`, soit dès que `i = fin + 1`, ou encore dès que `i > fin` puisque `i` augmente. Par conséquent, ce n'est pas fini tant que `i ≤ fin`.

Sinon, il nous faut réduire la distance entre l'état actuel et l'état final.

Se rapprocher de l'état final

On se rapproche de l'état final en augmentant le nombre de cases additionnées. Pour cela, il suffit d'en additionner une de plus. On obtient donc :

```
...
# Résultat = somme(t, début, i - 1)
Résultat <- Résultat + t[i]
# Résultat = somme(t, début, i)
i <- i + 1
# Résultat = somme(t, début, i - 1)
...
```

Les commentaires utilisent la fonction **somme** pour préciser l'état intermédiaire obtenu.

Initialiser le calcul

Initialiser le calcul, c'est placer le système logiciel dans un état dans lequel **(H)** est vraie. Elle énonce que toutes les cases, jusqu'à celle de numéro `i - 1` incluse, ont été additionnées. Par conséquent, elle énonce que la prochaine case à additionner est celle de numéro `i`. Or, initialement, la prochaine case à additionner est celle de numéro `début` et le résultat est encore nul :

```
variable
```


© ENI Editions - All rigths reserved

```
    i : ENTIER # Numéro de la prochaine case à additionner.
Initialisation
    i <- début
    Résultat <- 0
    # i = début => Résultat = 0
    # i > début => Résultat = somme(t, début, i - 1)
...
```

Il reste à arranger tout cela.

Rédiger l'algorithme définitif

L'hypothèse **(H)** s'exprime de différentes façons. Choisissons une expression algorithmique. C'est l'expression de l'invariant de l'itération :

```
...
invariant
    i = début => Résultat = 0
    i > début => Résultat = somme(t, début, i - 1)
...
```

D'autre part, nous devons aussi évaluer la distance qui sépare un état intermédiaire quelconque de l'état final. Nous avons vu que « c'est fini dès que i = fin + 1 ». Par conséquent, « c'est fini dès que fin - i + 1 = 0 ». De plus, `i` est croissant et, par conséquent, `fin - i + 1` est décroissant strictement vers 0. On obtient :

```
...
variant de contrôle
    fin - i + 1
...
```

L'algorithme définitif est alors le suivant :

*Algorithme 21 : Définition de la fonction **somme***

```
Algorithme somme
    # La somme des composantes de t[début .. fin].
Entrée
    t : TABLEAU[RÉEL]   # La cible du calcul de la somme.
    début, fin : ENTIER # Num. composantes extrêmes à additionner.
Résultat : RÉEL
précondition
    # début et fin sont des numéros de cases valides.
    index_valide(t, début)
    index_valide(t, fin)
    début ≤ fin
    # Les composantes de t[début .. fin] ont été initialisées.
    (∀k, début ≤ k ≤ fin)(t[k] ≠ NUL)
variable
    i : ENTIER # Numéro de la prochaine case à additionner.
Initialisation
    i <- début
    Résultat <- 0
    # i = début => Résultat = 0
    # i > début => Résultat = somme(t, début, i - 1)
tant que
    i ≤ fin
    invariant
        i = début => Résultat = 0
        i > début => Résultat = somme(t, début, i - 1)
    variant de contrôle
        fin - i + 1
répéter
    # i > début => Résultat = somme(t, début, i - 1)
    Résultat <- Résultat + t[i]
    # i > début => Résultat = somme(t, début, i)
```

© ENI Editions - All rigths reserved

```
    i <- i + 1
    # i > début => Résultat = somme(t, début, i - 1)
fin répéter
fin somme
```

Les commentaires placés entre **répéter ... fin répéter** montrent bien que l'invariant est rétabli et donc maintenu dans le corps de l'itération. Les traitements reprennent au début de l'itération parce que la situation, dans laquelle se trouve alors le système logiciel, est la même qu'à l'entrée de cette itération. L'expression de l'invariant permet de s'en assurer. De même, on vérifie que le variant de contrôle diminue à chaque pas.

On voit souvent exprimée une telle construction, dans laquelle toutes les cases sont parcourues, de la première à la dernière sans exception, d'une autre façon :

```
...
pour tout i de début à fin répéter
    additionner la composante de numéro i
...
```

Cette expression permet d'exprimer des itérations « automatiques », dans lesquelles la permanence de l'invariant est maintenue implicitement. Ainsi, la modification de l'indice `i` qui contrôle l'itération et le test de la condition d'arrêt sont implicites. L'exemple précédent s'écrirait alors :

```
...
pour tout i de début à fin répéter
    Résultat <- Résultat + t[i]
...
```

Ce n'est pas une bonne construction algorithmique car elle ne permet pas de suivre d'une façon consciente, raisonnée et explicite tous les états du système. Elle imite les constructions de certains langages de programmation mais elle nous semble plus nuisible qu'utile dans une phase d'apprentissage. Elle ne sera pas utilisée dans ce livre.

Ici encore, on peut penser que nous avons fait beaucoup d'efforts pour résoudre des problèmes aussi simples que le calcul de la somme ou de la moyenne des composants d'un tableau. Mais il faut se souvenir que cette discussion est motivée par une **solution fausse** à ces problèmes "simples". Ce qui montre que ces problèmes ne sont, dans le fond, pas aussi simples qu'ils le paraissent. Les efforts consentis sont donc ainsi justifiés.

Cette remarque termine l'étude à l'origine de cette discussion. Nous avons calculé la moyenne arithmétique des composantes d'un tableau. Pour effectuer le calcul effectif de la moyenne des composantes du tableau `marques`, il suffira de faire appel à cet algorithme en veillant à satisfaire à la précondition :

```
...
m <- moyenne(marques, n)
...
```

et c'est tout. Le reste n'est plus de l'algorithmique : c'est de la programmation. Pourtant, supposons que l'on souhaite, à toute force, écrire un algorithme qui remplit les cases d'un tableau `marques` par des opérations d'entrée/sortie, en comptant les composantes introduites.

3. Compléter l'exercice : les spécifications qui manquent

Le tableau a été déclaré avec deux valeurs fixées pour les numéros des composantes extrêmes, par exemple :

```
...
variable
    marques : TABLEAU[RÉEL][-3, 96]
...
```

On a ainsi déclaré un tableau qui contiendra des nombres réels et dont les cases sont numérotées de -3 à 96. Ces valeurs sont connues. La fonction **index_min** rend - 3 avec `marques` en paramètre. La fonction duale **index_max** rend 96. Le problème consiste donc à remplir `marques[-3 .. 96]` par des nombres réels. Soit donc **remplir** l'algorithme qui aura la responsabilité de cette initialisation. Est-ce une procédure ou une fonction ? C'est une décision importante : encore une fois, *une fonction ne modifie pas ses paramètres*. Elle reçoit des opérandes. Elle calcule et retourne un résultat : c'est une *requête*. Elle interroge le système sans provoquer d'*effet de bord*. L'algorithme à écrire reçoit en paramètre le tableau `marques` qu'il initialise. Par conséquent, il le modifie. De même, il reçoit un paramètre entier `n` qu'il initialisera avec le nombre de composantes qu'il aura placées dans `marques`. Ainsi, cet algorithme doit calculer deux résultats : un tableau qu'il remplit et un entier qu'il initialise. Si on en reste aux structures simples, cet algorithme est nécessairement une procédure. Il modifie l'état du système logiciel : c'est une *commande*. Là encore, on ne peut pas objecter que dans tel ou tel langage de programmation, une fonction peut modifier ses paramètres :

© ENI Editions - All rigths reserved

- nous ne programmons pas, nous analysons un problème ;
- nous ne disposons pas d'un langage pour cela ;
- ce n'est pas parce que tel ou tel langage permet n'importe quoi que nous sommes obligés de répéter les mêmes erreurs ;
- nous ne sommes pas tenus aux respects d'impératifs dictés par des machines, comme le sont les langages de programmation des machines réelles.

L'algorithme remplit le tableau `marques` entre les composantes de numéros **index_min**(marques) et **index_max** (marques) par des réels positifs. On peut introduire deux nouveaux paramètres entiers, `début` et `fin`, qui recevront respectivement le numéro de la première case et le numéro de la dernière case du tableau à initialiser. La signature de **remplir** devient :

```
Algorithme remplir
    # Initialiser des composantes de t[début .. fin] et les compter.
Entrée
    t : TABLEAU[RÉEL]    # Tableau cible de l'initialisation.
    début, fin : ENTIER # Numéros des cases extrêmes à initialiser.
    n : ENTIER # Nombre de cases effectivement initialisées.
```

Les paramètres `t` et `n` sont modifiés par la procédure. S'ils ont déjà des valeurs au début, elles ne sont pas utilisées et elles sont perdues au retour. Les paramètres `début` et `fin` ont des valeurs utilisées mais non modifiées par l'algorithme.

La précondition précise les pré-requis pour utiliser les services de l'algorithme :

```
...
précondition
    début ≤ fin
    index_valide(t, début)
    index_valide(t, fin)
...
```

Pour noter la postcondition, on définit un prédicat **estRempli** qui permettra d'exprimer que **remplir** fait bien ce qui lui est demandé. Il est donc possible de spécifier complètement **remplir** et cette spécification aura un sens dès que **estRempli** en aura un. L'algorithme suivant donne les spécifications de **remplir**.

*Algorithme 22 : Spécifications de **remplir***

```
Algorithme remplir
    # Initialiser n réels positifs dans t[début .. fin].
Entrée
    t : TABLEAU[RÉEL]    # Le tableau à initialiser.
    début, fin : ENTIER # Le domaine d'initialisation.
    n : ENTIER # Nombre de composantes effectivement initialisées.
précondition
    début ≤ fin
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    # début et fin ne sont pas modifiés.
    ancien(début) = début
    ancien(fin)   = fin
    # n composantes sont initialisées.
    début + n - 1 ≤ fin
    # t[début .. début + n - 1] est initialisé.
    estRempli(t, début, début + n - 1)
    # t[début .. début + n - 1] est initialisé avec des réels
    # positifs.
    (∀i, début ≤ i ≤ début + n - 1)(t[i] ≥ 0)
    # Le reste du tableau n'est pas modifié.
    (∀i, index_min(t) ≤ i ≤ début, début + n ≤ i ≤ index_max(t))
```

© ENI Editions - All rigths reserved

```
    (ancien(t[i]) = t[i])
fin remplir
```

Cette spécification précise clairement que `n` contient le nombre de cases initialisées et non pas le numéro de la dernière case initialisée. On aurait pu obtenir une autre solution en choisissant de renvoyer, dans `n`, le numéro de la dernière case initialisée. Cette adaptation est laissée en exercice.

La postcondition fait appel au prédicat **estRempli** en lui passant en paramètre le tableau et les numéros des cases extrêmes à vérifier. Pour que cette définition soit complète, il faut encore spécifier ce prédicat. C'est fait par l'algorithme suivant, dans lequel la postcondition est exprimée en français pour ne pas compliquer l'exercice.

*Algorithme 23 : Spécifications de **estRempli***

```
Algorithme estRempli
    # t[début .. fin] a-t-il été initialisé ?
Entrée
    t : TABLEAU[ RÉEL]    # Le tableau initialisé.
    début, fin : ENTIER # Le domaine d'initialisation.
précondition
    début ≤ fin
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    Résultat = (t[début .. fin] a été initialisé par des réels
    positifs)
fin estRempli
```

Il ne reste plus qu'à réaliser l'algorithme **remplir**. Cette fois, il faut analyser complètement le problème. On ne peut pas s'en sortir en écrivant directement le corps de l'algorithme. Si c'était le cas, on pourrait passer directement au codage dans le langage de programmation de la machine réelle.

4. Compléter l'exercice : remplir le tableau

Il s'agit d'interroger l'utilisateur pour lui demander un nombre et le ranger en bonne place. Ce dernier signale qu'il a terminé la saisie en introduisant un nombre convenu, qui ne fait pas partie du domaine des données valides. Le problème de départ était « résolu » comme on l'a vu avec la valeur -1. Comme il s'agit de n'introduire que des réels positifs ou nuls, on peut choisir de terminer sur une valeur négative mais non nulle quelconque. Continuons donc avec -1. Nous pouvons faire une hypothèse sur un état intermédiaire dans lequel certaines valeurs ont déjà été saisies.

Faire un hypothèse sur l'état actuel

Hypothèse (H) : n est le nombre de cases de t déjà initialisées entre début et i - 1. Le prochain réel à ranger est nombre.

On peut déjà remarquer que « les cases de t entre début et i - 1 sont déjà initialisées » s'écrit **estRempli**(t, début, i - 1) dès que `i > début`. Une deuxième remarque permettrait de simplifier **(H)** : `i` n'est pas nécessaire puisque `n` donne le nombre de cases déjà initialisées. Oublions cette remarque pour l'instant. Nous y reviendrons plus bas. Dans quelles circonstances **(H)** décrit-elle l'état final ?

Voir si c'est fini

C'est fini dès que `nombre < 0`. Mais c'est aussi fini si le tableau est plein, donc lorsque `i - 1 = fin`. On obtient :

```
...
C'est fini dès que nombre < 0 ou sinon  i - 1 = fin.
...
```

`i - 1 = fin` s'écrit aussi `fin - i + 1 = 0`. `fin - i + 1` est donc le variant de contrôle de l'itération. C'est fini dès que `i = fin + 1`, soit dès que `i > fin`. Sinon, il faut se rapprocher de l'état final.

Se rapprocher de l'état final

© ENI Editions - All rigths reserved

On s'en rapproche en rangeant `nombre` à sa place. Comme `i - 1` est le numéro de la dernière case initialisée d'après **(H)**, `nombre` doit être placé dans la case de numéro `i` :

```
...
assertion
    i > début => estRempli(t, début, i - 1)
    nombre est la prochaine valeur à ranger
    n est le nombre de cases déjà initialisées

t[i] <- nombre

assertion
    i > début => estRempli(t, début, i)
    nombre est la dernière valeur rangée
    n + 1 est le nombre de cases déjà initialisées
...
```

Un bloc introduit par **assertion** décrit un état par une conjonction de clauses de valeur booléennes. Le dernier bloc décrit l'état intermédiaire obtenu. Ce n'est pas celui qui est décrit par **(H)**. On le retrouve par :

```
...
i <- i + 1
n <- n + 1

assertion
    i > début => estRempli(t, début, i - 1)
    nombre est la dernière valeur rangée
    n est le nombre de cases déjà initialisées
```

Ce n'est pas encore ce que nous voulons. On doit obtenir, de plus :

```
...
    nombre est la prochaine valeur à ranger
...
```

Pour cela, il suffit de demander à l'utilisateur un nouveau nombre :

```
...
    écrire(ÉCRAN, 'Donner le nombre suivant : ')
    nombre <- lire(CLAVIER)
...
```

Remarquez que, là encore, la distinction est faite entre une procédure, **écrire**, qui est une commande envoyée au système et une fonction, **lire**, qui est une requête renvoyant un résultat. D'autre part, on a précisé les noms internes de fichiers à accéder, ici ÉCRAN en sortie et CLAVIER en entrée. Nous les supposons définis.

Il faut encore initialiser le calcul, c'est-à-dire réaliser **(H)** dans l'état initial.

Initialiser le calcul

La prochaine case à initialiser porte le numéro `i`. Initialement, aucune case n'a encore été remplie et la prochaine case à initialiser est donc la case de numéro `début`. La variable `n` contient le nombre de cases déjà initialisées et donc `n` est initialisé à 0. Enfin, `nombre` contient le prochain nombre à ranger dans `t`.

```
...
initialisation
    i <- début
    n <- 0
    écrire(ÉCRAN, 'Donner le premier nombre : ')
    nombre <- lire(CLAVIER)
...
```

Rédiger l'algorithme définitif

Il s'agit de rassembler tout cela. On obtient l'algorithme suivant, dans lequel les spécifications ne sont pas répétées.

*Algorithme 24 : Réalisation de **remplir***

© ENI Editions - All rigths reserved

```
Algorithme remplir
...
variable
    i      : ENTIER # Numéro de la prochaine case à initialiser.
    nombre : RÉEL   # Prochaine valeur à ranger dans t[i].
initialisation
    i <- début
    n <- 0
    écrire(ÉCRAN, 'Donner le premier nombre : ')
    nombre <- lire(CLAVIER)
jusqu'à
    nombre < 0
  ou sinon
    i > fin
    invariant
        i > début => estRempli(t, début, i - 1)
    variant de contrôle
        fin - i + 1
répéter
    assertion
        i > début => estRempli(t, début, i - 1)
        nombre est la prochaine valeur à ranger
        n est le nombre de cases déjà initialisées

    t[i] <- nombre

    assertion
        i > début => estRempli(t, début, i)
        nombre est la dernière valeur rangée
        n + 1 est le nombre de cases déjà initialisées
    i <- i + 1
    n <- n + 1
    écrire(ÉCRAN, 'Donner le nombre suivant : ')
    nombre <- lire(CLAVIER)

    assertion
        i > début => estRempli(t, début, i - 1)
        nombre est la prochaine valeur à ranger
        n est le nombre de cases déjà initialisées
fin répéter
...
fin remplir
```

Cet algorithme peut encore être amélioré. On remarque, par exemple, que, d'une part, `n` n'est jamais utilisé. D'autre part, `i` et `n` progressent ensemble, par incréments de 1, le premier depuis `début`, l'autre depuis 0. C'est signe que ce sont deux valeurs fortement liées : la valeur de l'une doit permettre d'obtenir la valeur de l'autre. Initialement, `i = début` et `n = 0`. Par conséquent, `i - n = début`. Comme les deux variables progressent ensemble, d'une unité à chaque pas de l'itération, cette différence reste constante. Après `k` pas de l'itération, on obtient :

```
(i + k) - (n + k) = début = i - n
```

Cette remarque permet alors de modifier l'algorithme de différentes façons :

(i)

supprimer `i` partout où il apparaît et remplacer toutes ses occurrences par sa valeur `début - n` ;

(ii)

supprimer la variable `n` partout et ajouter, après **fin répéter**, l'instruction qui calcule sa valeur : `n <- i - début`.

La solution (i) est correcte, mais elle fait trop de calculs inutiles. Elle ne sera pas retenue. L'écriture définitive de l'algorithme après adoption de la solution (ii) est laissée en exercice.

Il reste à dire comment utiliser tout cela, mais c'est une formalité :

© ENI Editions - All rigths reserved

```
...
variable
    marques : TABLEAU[ RÉEL] [- 3, 200]
    m : RÉEL
    n : ENTIER
...
# Initialiser le tableau avec au plus 100 réels positifs.
remplir(marques, 1, 100, n)
# Calculer et écrire la moyenne des nombres.
m <- moyenne(marques, n)
écrire(ÉCRAN, 'La moyenne est ', m)
...
```

Concluons cette section en précisant qu'un chapitre ultérieur définit un type **VECTEUR** et calcule la moyenne des composantes d'un vecteur.

© ENI Editions - All rigths reserved

Exercices d'application

Cette section propose des exercices d'application variés. Le premier exercice étend l'étude commencée au chapitre Structures élémentaires.

Exercice 5 : Historique sur un compte courant

On veut conserver l'histoire des mouvements mensuels sur un compte courant.

1. Modifier le type COMPTE défini au chapitre Structures élémentaires pour garder la trace des mouvements sur un compte pendant un mois.

2. Établir le solde en fin de mois d'un compte donné.

Le solde n'est plus un attribut du compte. Il est obtenu en parcourant l'historique mensuel et l'historique annuel qui enregistre le montant du solde du compte pour chaque mois.

3. Refaire les définitions précédentes.

Un tableau clients contient les comptes courants d'un ensemble de clients.

4. Déterminer les clients pour lesquels la moyenne des montants des mouvements est supérieure à une somme donnée.

Le problème de l'édition d'un entier a déjà été défini au chapitre Récursivité. L'exercice sera complètement résolu dans un chapitre ultérieur. L'exercice suivant propose d'aller un peu plus loin qu'au chapitre précédent.

Exercice 6 : Édition d'un entier

1. Écrire un algorithme itératif qui transforme un entier n quelconque en sa représentation dans une base B ≥ 2 quelconque.

Lorsque la base est supérieure à 36, on peut utiliser la représentation des chiffres en utilisant la base dix, mais en séparant chaque chiffre de la représentation du nombre en base B par un séparateur convenu, par exemple un point. C'est ce qui est pratiqué pour noter, par exemple, l'adresse IP d'un hôte sur un réseau en IPv4. Ainsi, par exemple, la représentation en base 256 de 3000, exprimée ici en base dix, s'écrira : $11.184_{256} = 3000_{dix}$.

Exercice 7 : Analyse d'une chaîne de caractères

On donne une chaîne de caractères dont différentes parties sont séparées par un caractère SÉPARATEUR particulier. L'exemple ci-dessous représente une telle chaîne, dans laquelle le caractère séparateur est le caractère deux-points ':'.

```
Voici un:exemple : de chaîne à analyser:
```

On veut séparer les différentes parties et les ranger dans un tableau de chaînes de caractères. Pour l'exemple ci-dessus, on veut obtenir le tableau suivant :

n°	***Chaîne***
1	'Voici un'
2	'exemple '
3	' de chaîne à analyser'

1. Écrire le module logiciel qui réalise cette décomposition.

2. Utiliser ce module dans un algorithme de recherche des champs GCOS du fichier `/etc/passwd` d'un système UNIX.

Il s'agit de construire un tableau donnant tous les champs obtenus associés au login d'un utilisateur. L'algorithme émet une alerte lorsque le champ GCOS n'est pas renseigné avec la description de l'utilisateur.

Exercice 8 : Recherche de mots dans un dictionnaire

*On donne un tableau de **MOTs** de la langue française. Les mots sont enregistrés dans un tableau appelé dictionnaire. Deux autres tableaux sont utilisés pour parcourir le dictionnaire. Un tableau appelé suivant donne, pour chaque mot, le numéro de la case occupée par le mot qui le suit dans l'ordre alphabétique dans le dictionnaire. Le tableau précédent donne de même, pour chaque mot, le numéro de la case du dictionnaire qui contient le mot qui le précède dans l'ordre alphabétique. Au premier mot de la liste triée correspond un numéro de case du précédent égal à **index_min**(dictionnaire) - 1. De même, au dernier mot de la liste triée correspond un numéro de case du suivant égal à la même valeur. Le tableau ci-dessous montre un*

© ENI Editions - All rigths reserved

exemple de ces trois tableaux.

i	***dictionnaire***	***précédent***	***suivant***
1	avion	3	4
2	train	4	...
3	auto	0	1
4	camion	1	2
5	...	...	...

On voit que, pour 'camion' dans la case 4, le mot qui le suit dans l'ordre alphabétique occupe la case 2 et celui qui le précède occupe la case 1. L'ordre alphabétique de ces trois mots est donc 'avion, 'camion' et 'train'.

1. Écrire l'algorithme qui donne la liste des mots enregistrés commençant par une lettre donnée.

Ainsi, par exemple, l'algorithme appliqué à la liste précédente donnera ('avion', 'auto') pour la lettre 'a'.

*Il est aussi possible de définir un nouveau type de données, **MOT**, formé d'une chaîne de caractères pour le mot proprement dit et de deux entiers pour repérer le mot qui le précède et le mot qui le suit dans le dictionnaire. Dans ce cas, le dictionnaire devient un simple tableau de **MOTs**.*

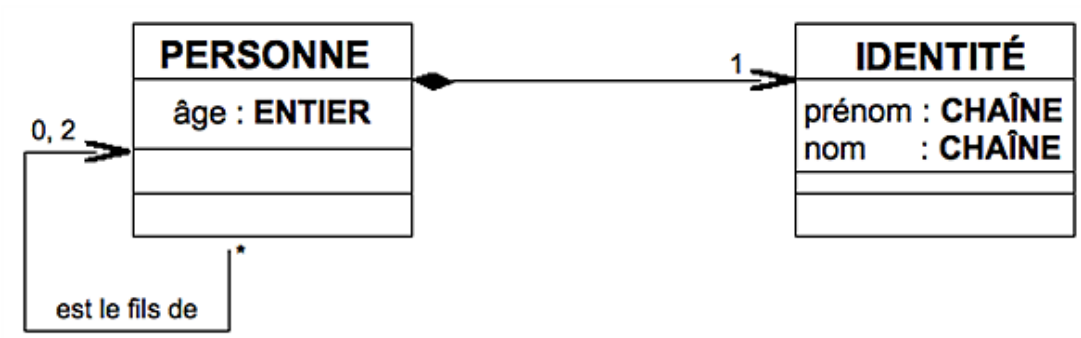
2. Définir le type MOT et refaire l'algorithme précédent pour l'utiliser.

3. Écrire l'algorithme qui permet d'ajouter un nouveau mot au dictionnaire.

4. Écrire l'algorithme de suppression d'un mot.

Exercice 9 : Représenter les membres d'une famille

*On a constitué un tableau familles de 1000 composantes numérotées à partir de 1. Chaque composante contient une donnée de type **PERSONNE** selon le diagramme suivant :*



*L'association réflexive entre **PERSONNE** et elle-même, désignée sur le diagramme par « est le fils de » signifie qu'une instance de **PERSONNE** est fille (fils) d'aucune, d'une ou de deux instances de **PERSONNE**.*

*1. Définir le type **PERSONNE**. Comment déclarer le tableau familles ?*

Lorsqu'une personne enregistrée dans le tableau n'a pas de père ou de mère enregistré, l'attribut correspondant est ORPHELIN. Lorsqu'une case est inoccupée parce que la personne qu'elle contenait a été effacée, son identifiant est EFFACÉ. Les cases qui n'ont jamais reçu de valeur, lorsqu'il y a moins de 1000 personnes enregistrées, ont un identifiant égal à VIDE.

2. Donner la liste de toutes les personnes enregistrées, âgées de 20 à 30 ans.

3. Vieillir toutes les personnes enregistrées de 1 an.

4. Établir la liste de tous les orphelins de moins de 15 ans.

5. Faire un algorithme qui détermine l'identité du père de 'Jacques MARTIN'.

6. Faire un algorithme qui établit la liste des frères et sœurs de 'Jacques MARTIN'.

Exercice 10 : PGCD de deux entiers

L'algorithme qui permet de déterminer le Plus Grand Commun Diviseur de deux entiers a déjà été envisagé au chapitre

© ENI Editions - All rigths reserved

*Récursivité, lors de l'étude sur le type **FRACTION**.*

1. Donner une version itérative de l'algorithme d'Euclide pour le calcul du PGCD de deux entiers.

2. Étudier une version itérative qui calcule le PGCD en ne faisant que des additions ou des soustractions.

Exercice 11 : Carrés parfaits et racine carrée entière

Un entier naturel est un carré parfait, s'il est le carré d'un entier. Ainsi, $0 = 0^2$, $1 = 1^2$, $4 = 2^2$, $16 = 4^2$ sont des carrés parfaits.

1. Faire un algorithme qui établit la liste des carrés parfaits inférieurs à une limite donnée. L'algorithme ne doit pas faire de multiplication.

La racine carrée entière d'un entier $n \geq 0$ est l'unique entier $r \geq 0$ défini par :

$r^2 \leq n < (r + 1)^2$

2. Écrire l'algorithme de calcul de la racine carrée entière d'un entier. Faire deux versions : récursive et itérative.

© ENI Editions - All rigths reserved

Notes bibliographiques

La méthode de construction d'une itération par étape, toujours les mêmes, comme elle est exposée ici doit beaucoup à [ARS80]. Le livre [ARS77] est de la même veine, mais adopte un point de vue plus mathématique et plus formel. Préciser l'invariant et le variant de contrôle d'une itération sous une forme algorithmique a été systématisé par Bertrand MEYER dont la référence [MEY00], qui traite de conception et programmation objet, est un point d'entrée pour aller plus loin.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a présenté la construction raisonnée d'une itération. Une itération est un traitement répété sur un jeu de données qui évoluent, mais qui restent liées par des conditions strictes. Construire l'itération, c'est énoncer clairement l'hypothèse de récurrence sous-jacente pour en déduire les traitements qui assurent les changements d'état du système, les initialisations qui réalisent l'état initial, la condition de terminaison qui décrit l'état final, l'invariant qui énonce une propriété dont les traitements, dans tous les états du système, assurent la permanence et le variant de contrôle qui est une mesure de la distance à l'état final. Les exemples proposés ont montré comment construire une itération selon ce « modèle ». Ils sont empruntés à différents domaines mais ne font jamais appel à d'autres structures de données que le tableau simple.

Les chapitres suivants ne sont que des variations sur le thème de la construction d'une itération et de la spécification récursive d'un algorithme.

© ENI Editions - All rigths reserved

Bibliographie

[ARS80] Jacques ARSAC : *Premières leçons de programmation* ; CEDIC/NATHAN, 1980.

[ARS77] Jacques ARSAC : *La construction de programmes structurés* ; DUNOD, 1977.

[MEY00] Bertrand MEYER : *Conception et programmation orientées objet* ; EYROLLES, 2000.

© ENI Editions - All rigths reserved

Introduction

Le chapitre Récursivité a commencé à montrer comment définir un module qui réalise un traitement répété, en utilisant ses propres services. Nous avons aperçu que la réalisation d'un tel module exprime une définition, au sens mathématique du terme, plutôt qu'une suite de transformations à appliquer aux données. Le chapitre Itération a étudié, en détail, comment raisonner sur un problème qui transforme les données par un traitement répété. Ce court chapitre réalise une synthèse sur les traitements répétés en comparant les deux méthodes.

La section Retour sur la récursivité revient sur la récursivité en présentant plusieurs exemples élémentaires mais significatifs de récursivité simple. La section Récursivité ou itération ? donne quelques éléments pour comparer récursivité et itération et met en évidence les raisons pour lesquelles l'itération est préférée à la récursivité pour la programmation de traitements répétés. La section Exercices, enfin, propose quelques exercices. Le chapitre se termine par un résumé.

© ENI Editions - All rigths reserved

Retour sur la récursivité

Un algorithme récursif exprime donc une définition au lieu de dire comment réaliser les transformations sur les données. Revoyons ce point sur un exemple élémentaire, déjà résolu au chapitre Récursivité.

1. Premier exemple

Rappelons d'abord le problème.

Un caractère est un nombre entier positif ou nul. Dans la suite, un caractère est donc confondu avec son code numérique. Le domaine des valeurs d'un tel entier n'est pas précisé davantage. On peut y penser en supposant, par exemple, que le domaine dépend de la machine utilisée. Le type de données **CARACTÈRE** permet de définir une variable de type caractère.

L'un de ces caractères est un code particulier. Il est désigné par FIN_CH. Ce caractère *n'est jamais un caractère significatif* dans une chaîne. Il n'est utilisé que pour marquer la terminaison d'une chaîne. Ainsi, toute donnée de type **CHAÎNE** contient, comme dernier caractère, celui de code FIN_CH qui en marque la fin. Les caractères d'une chaîne sont numérotés en séquence à partir de 1 pour le premier.

L'accesseur **item**, défini précédemment, reste disponible dans le répertoire d'instructions. C'est une fonction qui prend en entrée une chaîne de caractères bien formée, c'est-à-dire terminée par FIN_CH, et un entier naturel rang. Elle rend le caractère de la chaîne qui occupe le rang donné, comme dans les chapitres précédents. De même, les fonctions **premier** et **fin** définies dans les chapitres précédents restent disponibles. La fonction **premier** rend une copie du premier caractère de la chaîne qu'elle reçoit en paramètre. La fonction **fin** retourne une copie de la chaîne qu'elle reçoit en paramètre, mais privée de son premier caractère.

On s'intéresse au calcul du nombre de caractères d'une chaîne. Le caractère FIN_CH n'est pas compté.

Le problème est donc de *définir un algorithme qui rend le nombre de caractères qui composent une chaîne bien formée*. Il existe plusieurs façons de résoudre ce problème.

La plus simple et immédiate consiste à donner une définition de la longueur de la chaîne. Comme la chaîne est bien formée par hypothèse, nous sommes certains d'y trouver le caractère FIN_CH. La longueur d'une chaîne réduite au seul caractère FIN_CH est nulle :

```
longueur('FIN_CH') = 0
```

Autrement dit, la chaîne est de longueur nulle si son premier caractère est la marque de fin de chaîne. Soit ch la chaîne de caractères bien formée dont on calcule la longueur. On obtient :

```
premier(ch) = FIN_CH => longueur(ch) = 0
```

Dans le cas contraire, cette longueur est celle de la chaîne, privée de son premier caractère, mais augmentée d'une unité :

```
premier(ch) ≠ FIN_CH => longueur(ch) = 1 + longueur(fin(ch))
```

La définition de la longueur d'une telle chaîne de caractères bien formée est donc :

```
premier(ch) = FIN_CH => longueur(ch) = 0
premier(ch) ≠ FIN_CH => longueur(ch) = 1 + longueur(fin(ch))
```

L'expression algorithmique de cette définition devient :

Algorithme 1 : longueur d'une chaîne de caractères - Version 1.0

```
Algorithme longueur
    # Le nombre de caractères de ch.
Entrée
    ch : CHAÎNE
Résultat : ENTIER
précondition
    ch ≠ NUL
    ch est bien formée
postcondition
    premier(ch) = FIN_CH => Résultat = 0
```

© ENI Editions - All rigths reserved

```
    premier(ch) ≠ FIN_CH => Résultat = 1 + longueur(fin(ch))
fin longueur
```

La réalisation de l'algorithme détaille les instructions à programmer dans un langage informatique donné. Avec les conventions des chapitres précédents, on écrirait :

```
réalisation
    si premier(ch) = FIN_CH
    alors Résultat <- 0
    sinon Résultat <- 1 + longueur(fin(ch)) fin si
```

Il est important de bien se convaincre que cette réalisation n'apporte rien de plus que l'algorithme précédent, du strict point de vue algorithmique. Elle ne fait qu'exprimer, dans un langage particulier, les instructions qu'il faudrait programmer dans un langage informatique, au besoin en les adaptant, pour implanter l'algorithme.

Le chapitre Itération a étudié comment construire une itération d'une façon raisonnée. Une itération détaille les transformations à faire subir aux données pour faire parcourir, au système logiciel, un espace d'états jusqu'à un état final bien défini. L'une des particularités d'une itération est de réaliser des transitions entre états caractérisés par une propriété invariante. La réalisation de la version itérative du calcul de la longueur d'une chaîne de caractères bien formée s'écrirait, par exemple :

```
réalisation
    Résultat <- 0 # Longueur actuelle de la chaîne.
    i <- 1        # Prochain caractère à observer.
    tant que
        item(ch, i) ≠ FIN_CH
    répéter
        Résultat <- Résultat + 1
        i <- i + 1
    fin répéter
```

On peut remarquer que `i =`**`Résultat`** `+ 1` et qu'il est donc possible d'éliminer `i` de cette solution, mais c'est un détail.

Cette version, bien que plutôt simple, est bien moins évidente à comprendre que la version récursive. Il faut faire un effort, même pour un problème aussi simple, pour comprendre comment l'algorithme calcule son résultat et comment il atteint l'état final. La version récursive est incomparablement plus expressive que la version itérative et c'est toujours le cas. C'est que la version récursive ne dit pas comment atteindre l'état final. Elle ne fait que donner une définition du résultat. La version itérative, au contraire, s'attache aux détails des transformations qui vont faire parcourir au système logiciel tous les états intermédiaires jusqu'à l'état final, dans lequel la pseudo-variable **Résultat** contient la valeur cherchée. Étudions un deuxième exemple.

2. Deuxième exemple

Voici la définition d'un algorithme **A**. *Que fait-il ?*

```
Algorithme A
    # ?
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
Résultat : BOOLÉEN
précondition
    car ≠ NUL
    ch  ≠ NUL et ch est bien formée
réalisation
    Résultat <- FAUX
    tant que
        ch ≠ CHAÎNE_VIDE et Résultat = FAUX
    répéter
        Résultat <- (premier(ch) = car)
        ch <- fin(ch)
    fin tant que
postcondition
...
fin A
```

La définition du résultat calculé est :

© ENI Editions - All rigths reserved

```
ch = CHAÎNE_VIDE => Résultat = FAUX
ch ≠ CHAÎNE_VIDE => Résultat =
                        (
                            premier(ch) = car
                          ou sinon
                            A(fin(ch), car)
                        )
```

Le résultat est donc FAUX pour une chaîne vide. Il est VRAI lorsque `car` est le premier caractère de la chaîne ou d'une sous-chaîne qui termine `ch`. Par conséquent, il est VRAI lorsque `car` est un caractère de `ch`.

La définition est immédiatement compréhensible. La version itérative demande plus de travail et d'intuition.

3. Troisième exemple

Considérons la réalisation suivante :

```
Entrée
    n : ENTIER
Résultat : ENTIER
précondition
    n ≥ 0
réalisation
    Résultat <- 1
    tant que
        n > 1
    répéter
        Résultat <- Résultat x n
        n <- n - 1
    fin répéter
postcondition
...
```

La définition correspondante est :

```
n ≤ 1 => Résultat = 1
n > 1 => Résultat = n x f(n - 1)
```

ce qui révèle immédiatement la fonction factorielle.

Puisque la récursivité est si expressive et immédiatement compréhensible, pourquoi faire encore l'effort d'étudier des solutions itératives ? Les langages informatiques appelés « langages fonctionnels » ne pratiquent pas l'itération et ne connaissent pas l'instruction d'affectation. Les solutions ne sont exprimées que récursivement. Pourquoi, dans ce cas, utiliser encore des langages impératifs et des algorithmes itératifs ? La section suivante donne une réponse partielle à cette question.

© ENI Editions - All rigths reserved

Récursivité ou itération ?

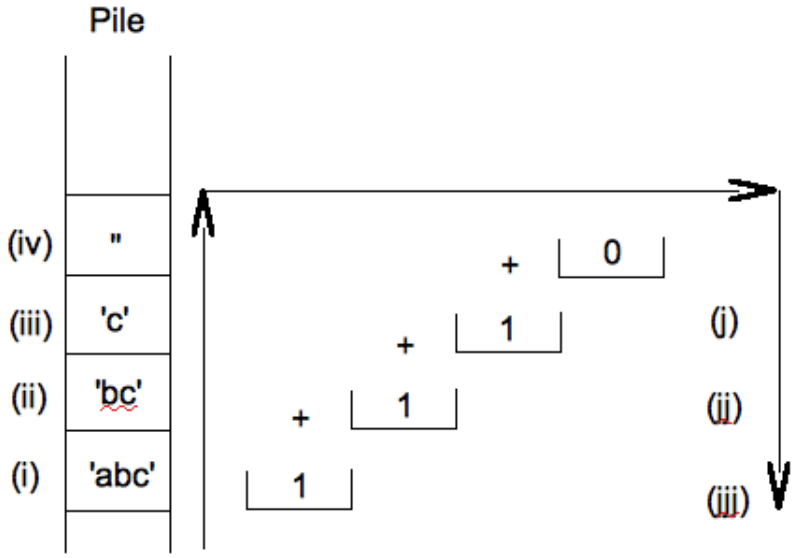
Pour comprendre la différence essentielle qui fait préférer un algorithme itératif à un algorithme récursif, étudions le « fonctionnement » des deux versions du calcul de la longueur d'une chaîne de caractères, comme il a été présenté à la section précédente.

Les étapes du calcul récursif sont les suivantes :

```
(i)  : 'abc' ≠ CHAÎNE_VIDE => longueur('abc') = 1 + longueur('bc') ;
(ii) : 'bc'  ≠ CHAÎNE_VIDE => longueur('bc')  = 1 + longueur('c')  ;
(iii): 'c'   ≠ CHAÎNE_VIDE => longueur('c')   = 1 + longueur('')   ;
(iv) : ''    = CHAÎNE_VIDE => longueur('')    = 0

(j)  : longueur('')   = 0  => longueur('c')   = 1 + 0 = 1 ;
(jj) : longueur('c')  = 1  => longueur('bc')  = 1 + 1 = 2 ;
(jjj): longueur('bc') = 2  => longueur('ab')  = 1 + 2 = 3
```

Les éléments successifs du calcul, c'est-à-dire les chaînes de caractères sur lesquelles opèrent l'algorithme, sont placés en attente du résultat intermédiaire pour terminer le calcul. Le calcul de la longueur de 'abc' est mis en attente du résultat de la longueur de 'bc'. Ce calcul est mis en attente du résultat du calcul de la longueur de 'c'. Ce dernier calcul est mis en attente du résultat du calcul de la longueur de '' qui est 0. Alors, la longueur de 'c' est calculée en ajoutant 1 au résultat déjà obtenu, ce qui donne une longueur de 'c' égale à 1. La longueur de 'bc' est alors calculée en ajoutant 1 à celle de 'c', ce qui donne 2. Enfin, la longueur de 'abc' est obtenue en ajoutant 1 à celle de 'bc' ce qui donne le résultat attendu.

Ce qui est remarquable ici, c'est que les calculs en attente imposent une sauvegarde de l'état intermédiaire du système logiciel qui doit être réutilisé ensuite pour terminer les calculs. Cette sauvegarde nécessite un espace de stockage et de nouveaux calculs intermédiaires pour sa gestion. La profondeur de l'espace de stockage et les calculs nécessaires à la gestion de cet espace mobilisent des ressources supplémentaires proportionnellement à la longueur de la chaîne. Le schéma de la figure ci-dessous montre comment évolue cet espace avec les différents états intermédiaires du système.



La partie gauche symbolise la pile qui sauvegarde les contextes temporaires pour mettre en attente les calculs. La partie centrale montre l'évolution de la valeur de **Résultat**. Les contextes sont ensuite dépilés pour calculer le résultat définitif.

Ce qui rend nécessaire l'empilement des contextes, c'est que le résultat est obtenu par un calcul qui fait intervenir un appel à la fonction :

```
Résultat <- 1 + longueur(fin(ch))
```

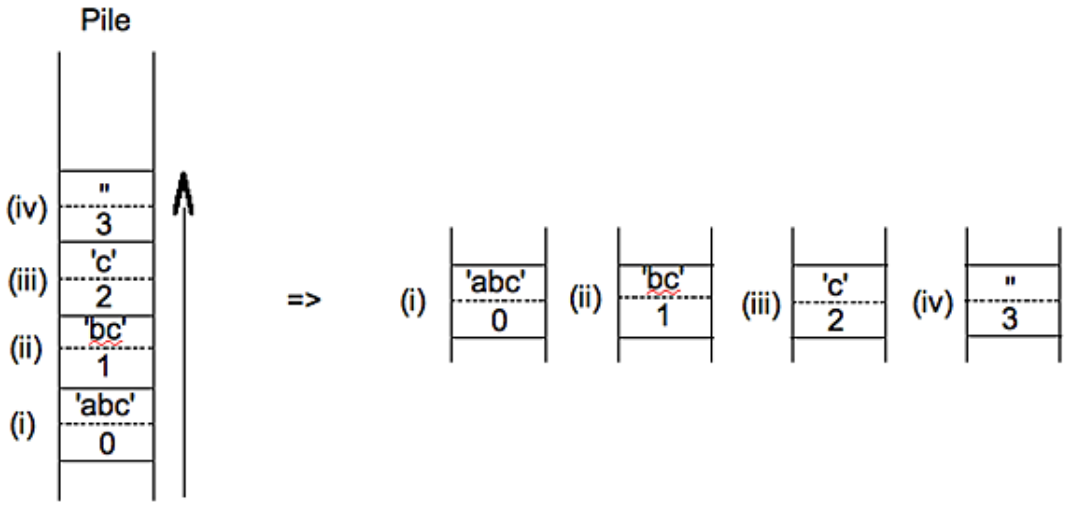
Pour calculer le résultat, il faut d'abord calculer la longueur de **fin**(ch), ce qui nécessite la sauvegarde provisoire de l'état actuel.

C'est la sauvegarde des contextes intermédiaires successifs qui fait parfois considérer la récursivité comme inefficace et c'est la raison pour laquelle l'itération lui est préférée. Pourtant, les spécialistes de la théorie des langages et les informaticiens ont développé des méthodes de transformations automatiques qui permettent de réduire les pénalisations induites par la récursivité. La sauvegarde du contexte reste nécessaire, au moins partiellement, mais il

© ENI Editions - All rigths reserved

est possible de procéder autrement que ci-dessus.

Supposons que soit empilé, avec la chaîne de caractères, la valeur actuelle du résultat. La pile évoluera alors comme le montre le dessin de la figure suivante :



La partie gauche du dessin montre la pile et les contextes avec lesquels sont sauvegardées les valeurs provisoires du résultat. La partie droite montre les éléments de pile dans les différents états. Comme seule importe la dernière position de pile qui contient le résultat définitif, il n'est pas nécessaire de maintenir les valeurs intermédiaires. Par conséquent, chacune des positions intermédiaires peut être écrasée par la nouvelle position, ce qui montre que la pile est, cette fois, restreinte à une seule position. L'entrée (ii) écrase l'entrée (i) et elle est à son tour écrasée par l'entrée (iii). Lorsque la chaîne est vide, comme ici à l'entrée (iv), le résultat définitif est obtenu.

Notons alors, en second paramètre de la fonction **longueur**, le résultat provisoire obtenu dans chaque état. On obtient :

```
longueur('abc', 0) = longueur('bc', 1) =
longueur('c', 2) = longueur('', 3)
```

ce qui termine le calcul.

Le second paramètre est appelé un *accumulateur*. La fonction **longueur** s'écrit :

Algorithme 2 : longueur d'une chaîne de caractères - Version 2

```
Algorithme longueur
    # Nombre de caractères de ch.
Entrée
    ch  : CHAÎNE # La chaîne dont on veut la longueur.
    ACC : ENTIER # L'accumulateur.
Résultat : ENTIER
précondition
    ch ≠ NUL
réalisation
    si
        ch = CHAÎNE_VIDE
    alors
        Résultat <- ACC
    sinon
        Résultat <- longueur(fin(ch), ACC+1)
    fin si
postcondition
...
fin longueur
```

Pour utiliser les services de cet algorithme, il suffit de l'appeler avec la chaîne de caractères à étudier et un accumulateur initialisé à 0.

```
...
```


© ENI Editions - All rigths reserved

```
phrase : CHAÎNE
nbCar  : ENTIER
...
initialiser phrase
nbCar <- longueur(phrase, 0)
...
```

Bien entendu, toute erreur dans l'initialisation de l'accumulateur provoque une erreur de calcul et un résultat faux renvoyé par la fonction.

La différence de comportement des deux fonctions vient de ce que la première fait appel à ses propres services dans une expression arithmétique. L'appel récursif participe à cette expression en tant qu'opérande. On parle alors de *récursivité non terminale*. Dans la seconde fonction, l'appel récursif intervient toujours, mais pas en tant qu'opérande d'une instruction de calcul. On parle alors de *récursivité terminale*. Lorsque la récursivité est terminale, il n'est pas nécessaire de sauvegarder les états intermédiaires du système puisque les valeurs intermédiaires de cet état ne participe plus au calcul du résultat.

Lorsque la récursivité terminale est exploitée en programmation, les compilateurs de langages informatiques modernes savent la reconnaître et la transformer automatiquement en itération. Autrement dit, l'étude de la solution algorithmique peut se satisfaire d'une solution récursive terminale dont nous avons vu qu'elle est plus expressive et plus facilement obtenue qu'une solution itérative. On laissera, au compilateur du langage d'implémentation, le soin de la transformer en itération pour gagner sur les performances.

La suite de ce chapitre propose quelques exercices pour s'entraîner à pratiquer l'analyse algorithmique et la récursivité pour des solutions où elle est terminale.

© ENI Editions - All rigths reserved

Exercices

*Exercice résolu 1 : Fonction **factorielle***

Transformer l'algorithme de la fonction factorielle étudiée au chapitre Récursivité pour en faire un algorithme dont la récursivité est terminale.

Solution

Il suffit d'utiliser un accumulateur initialisé à 1 puisque **factorielle**(0) = **factorielle**(1) = 1.

Algorithme 3 : Fonction factorielle (récursivité terminale)

```
Algorithme factorielle
    # n!
Entrée
    n   : ENTIER
    ACC : ENTIER
précondition
    n ≥ 0
réalisation
    si
        n < 2
    alors
        Résultat <- ACC
    sinon
        Résultat <- factorielle(n - 1, ACC x n)
    fin si
postcondition
    Résultat = n!
fin factorielle
```

La fonction est utilisée ainsi :

```
k <- factorielle(10, 1) # calcule 10!
```

*Exercice résolu 2 : Fonction **appartient***

*1. Écrire une fonction **appartient** qui rend VRAI si et seulement si le caractère qu'elle reçoit en second paramètre compose la chaîne qu'elle reçoit en premier paramètre.*

2. La solution proposée est-elle une récursivité terminale ?

Solution

Le résultat est évidemment FAUX lorsque la chaîne de caractères est vide :

```
ch = CHAÎNE_VIDE => Résultat = FAUX
```

Le résultat est VRAI si le caractère cherché est le premier de la chaîne quand elle n'est pas vide :

```
ch ≠ CHAÎNE_VIDE => Résultat = (premier(ch) = car) ...
```

Lorsque ce n'est pas le premier, le résultat est VRAI si le caractère appartient au reste de la chaîne :

```
... ou sinon Résultat = appartient(fin(ch), car))
```

L'algorithme demandé est donc :

*Algorithme 4 : Fonction **appartient***

© ENI Editions - All rigths reserved

```
Algorithme appartient
    # car appartient-il à ch?
Entrée
    ch  : CHAÎNE
    car : CARACTÈRE
précondition
    ch  ≠ NUL
    car ≠ NUL
réalisation
    si
        ch = CHAÎNE_VIDE
    alors
        Résultat <- FAUX
    sinon
        Résultat <- (
                        premier(ch) = car
                      ou sinon
                        appartient(fin(ch), car))
                    )
    fin si
postcondition
...
fin appartient
```

Exprimée ainsi, la récursivité n'est pas terminale puisque l'appel récursif à **appartient** compose une expression booléenne. Mais il n'est pas nécessaire d'utiliser un accumulateur ici, en refactorisant les instructions de l'alternative :

```
réalisation
    si
        ch = CHAÎNE_VIDE
    alors
        Résultat <- FAUX
    sinon si
        premier(ch) = car
    alors
        Résultat <- VRAI
    sinon
        Résultat <- appartient(fin(ch), car))
    fin si
```

L'exercice suivant est moins trivial.

*Exercice résolu 3 : Fonction **fibonacci***

Écrire une fonction récursive qui calcule l'élément de rang n de la suite de Fibonacci.

Solution

Nous avons déjà vu dans un chapitre antérieur que la suite de Fibonacci est définie par :

$$fibonacci(0) = fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2),\ n \geq 2$$

Une première solution consiste à paraphraser cette définition :

```
si
    n < 2
alors
    Résultat <- 1
sinon
    Résultat <- fibonacci(n – 1) + fibonacci(n – 2)
fin si
```

Cependant, les appels récursifs sont les deux opérandes d'une addition. Pour rendre terminale la récursivité, le contexte doit sauvegarder ainsi deux valeurs. Une solution consiste donc à utiliser deux accumulateurs F1 et F2 :

© ENI Editions - All rigths reserved

```
Entrée
    n, F1, F2 : ENTIER
Résultat : ENTIER
précondition
    n ≥ 0
réalisation
    si
        n < 2
    alors
        Résultat <- F1
    sinon
        Résultat <- fibonacci(n - 1, F1 + F2, F1)
    fin si
postcondition
...
```

La fonction est appelée par :

```
v <- fibonacci(50, 1, 1) # calcul du 50ème terme.
```

Exercice 1 : Addition de deux entiers

On donne deux entiers.

Calculer récursivement leur somme en ne réalisant que des incrémentations.

La soustraction n'est pas autorisée. Seules l'addition de 1 et l'alternative sont autorisées.

Exercice 2 : Puissance d'un réel par un entier

1. Calculer récursivement x^n.

La récursivité doit être terminale.

2. Transformer en un algorithme itératif puis programmer effectivement les solutions pour comparer les durées d'exécution sur des jeux de données identiques.

D'une façon générale, tous les exercices traités dans les chapitres Récursivité et Itération peuvent être repris pour obtenir une solution récursive ou récursive terminale. Énumérer les énoncés de ces exercices n'apporterait rien de plus ici.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre est une courte introduction aux problèmes soulevés par la récursivité. Nous avons vu quelques éléments permettant d'être sensibilisé aux inconvénients de la récursivité, mais aussi les avantages incomparables que présentent ces solutions, notamment dans la puissance d'expressivité qu'elles permettent. Nous avons introduit la récursivité terminale et précisé les raisons qui font que ces solutions peuvent être utilisées, en programmation, sans inconvénient majeur avec un compilateur de langage moderne. Il faudrait aussi montrer comment transformer les algorithmes récursifs en algorithmes itératifs et comment prouver un algorithme récursif, mais ces problèmes dépassent le cadre de cette initiation.

© ENI Editions - All rigths reserved

Introduction

Ce chapitre présente le tri de données comparables. On considère une collection C de données de même type **T**, quelconque mais **COMPARABLEs**. On veut obtenir une représentation ordonnée des éléments de la collection. Les données seront organisées le plus souvent en tableau. On ne cherche donc pas ici des tris de qualité, évalués selon leurs performances, mais plutôt un prétexte à la construction raisonnée d'algorithmes.

La section Spécifier un algorithme de tri commence par poser le problème. C'est la partie difficile du chapitre et elle peut être ignorée en première lecture. En particulier, la spécification algorithmique complète d'un algorithme de tri impose la définition des multi-ensembles invariables et de certaines opérations applicables difficiles à spécifier. La section suivante étudie quelques algorithmes usuellement définis dans une initiation à l'algorithmique. La section Fusionner deux tableaux triés montre comment fusionner deux tableaux triés. La section Exercices, enfin, propose des exercices plus difficiles.

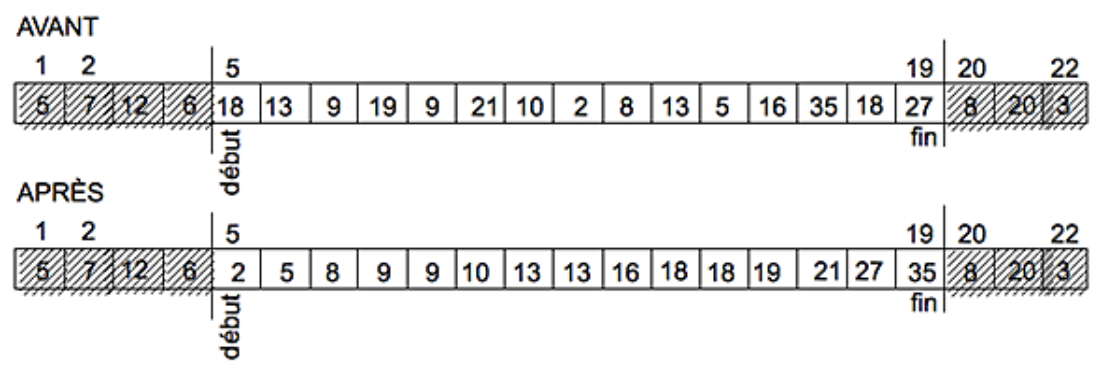
© ENI Editions - All rigths reserved

Spécifier un algorithme de tri

Cette section est faite de deux parties. La première présente le problème du tri de données comparables. La seconde étudie la postcondition d'un tri. C'est la partie vraiment difficile du chapitre et elle peut être ignorée en première lecture.

1. Présentation du problème du tri

Soit t un tableau dont les composantes sont d'un type **T** dérivé de **COMPARABLE**. On veut un algorithme qui replace dans t ses composantes en ordre, par exemple, croissant. La figure suivante représente un tableau d'entiers avant et après le tri.



Comme d'habitude, l'algorithme à définir intervient sur une partie précisée du tableau, entre les cases de numéros `début` et `fin`. Les données sont replacées en ordre croissant dans le même tableau, qui se trouve donc modifié par l'algorithme. Par conséquent, il s'agit d'écrire une procédure. Les données en entrée sont le tableau `t` et les numéros des composantes extrêmes `début` et `fin`. La signature de la procédure et sa précondition peuvent donc être précisées :

```
Algorithme trierAsc
    # Trier t[début .. fin] en ordre croissant.
Entrée
    t : TABLEAU[T -> COMPARABLE] # Le tableau à trier.
    début, fin : ENTIER # Numéros des composantes extrêmes à trier.
précondition
    # début et fin sont des index valides de t.
    index_valide(t, début)
    index_valide(t, fin)
    # t[début .. fin] est défini.
    estDéfini(t, début, fin)
```

L'algorithme **estDéfini** est un prédicat VRAI si et seulement si la partie `t[début .. fin]` a été initialisée.

*Algorithme 1 : Spécifications de **estDéfini***

```
Algorithme estDéfini
    # t[début .. fin] a-t-il été initialisé ?
Entrée
    t : TABLEAU[T -> COMPARABLE] # Le tableau à explorer.
    début, fin : ENTIER # Numéros composantes extrêmes à vérifier.
Résultat : BOOLÉEN
précondition
    # début et fin sont des index valides de t.
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    # FAUX lorsque les indices ne sont pas en ordre.
    début > fin => Résultat = FAUX
    # VRAI si et seulement si les composantes de t[début .. fin]
    # sont non nulles.
```

© ENI Editions - All rigths reserved

```
    début = fin => Résultat = (t[début] ≠ NUL)
    début < fin => Résultat =
        (
            (t[début] ≠ NUL)
          et alors
            estDéfini(t, début+1, fin)
        )
```

La postcondition du tri est plus difficile à obtenir. C'est elle qui est étudiée dans la section suivante.

2. Étude de la postcondition du tri

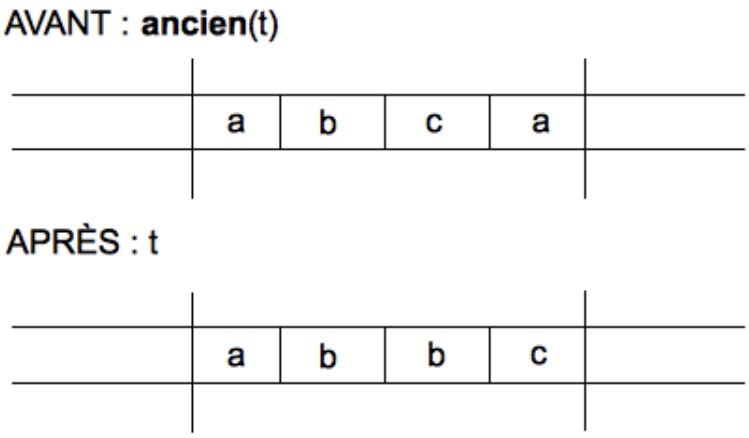
La postcondition précise d'abord que la partie `t[début .. fin]` du tableau `t` est triée, ici en ordre croissant. La clause qui exprime cet état a déjà été écrite au chapitre Itération. Elle utilise le prédicat **estTriéAsc** dont le chapitre Itération a étudié une version itérative. On obtient donc :

```
...
postcondition
    # début et fin ne sont pas modifiés.
    ancien(début) = début
    ancien(fin)   = fin
    # t[début .. fin] est trié en ordre croissant.
    estTriéAsc(t, début, fin)
...
```

Cependant, nous devons exprimer que la partie `t[début .. fin]` a certes été modifiée, mais que les mêmes éléments restent présents. Ainsi, les tableaux `t` et **ancien**(`t`) ont les mêmes éléments, dans les mêmes sous-tableaux, entre les cases de numéros `début` et `fin`, mais qu'ils ne sont pas nécessairement identiques puisque certaines composantes ont peut-être été déplacées. Nous ne pouvons pas écrire la condition (`c1`) :

```
(∀k, début ≤ k ≤ fin)(∃i, début ≤ i ≤ fin)(t[k] = ancien(t)[i])
```

car il peut exister des composantes de `t` en plusieurs exemplaires entre début et `fin`. Considérons, par exemple, les tableaux de la figure suivante :



Le résultat `t` est trié en ordre croissant et il passe le test exprimé par la condition (`c1`). Pourtant, le tri réalisé n'est pas correct puisqu'on ne retrouve pas dans `t[début .. fin]` tous les éléments de **ancien**(`t`)`[début .. fin]`. Le second exemplaire de l'élément `a` a été remplacé par un nouvel exemplaire de l'élément `b`. Nous devons pouvoir nous assurer que `t` et **ancien**(`t`) possèdent exactement le même nombre d'exemplaires des mêmes composantes. Pour définir un prédicat qui permettra de les comparer, on peut « éliminer », à chaque étape de la comparaison, les composantes trouvées dans les deux tableaux. Pour simplifier les notations, écrivons les composantes des tableaux en utilisant la notation ensembliste. Pour les tableaux de la figure ci-dessus, on a, initialement :

```
E = {a, b, c, a}
F = {a, b, b, c}
```

`E` et `F` sont appelés des multi-ensembles.

© ENI Editions - All rigths reserved

Définition

On appelle multi-ensemble un ensemble dans lequel les éléments peuvent apparaître en plusieurs exemplaires.

Pour vérifier que les deux multi-ensembles sont égaux, on compare les deux premiers éléments et on les trouve égaux. Ils sont éliminés et on obtient alors les deux multi-ensembles : {b, c, a} et {b, b, c}. Une nouvelle comparaison permet de constater que les premiers éléments de chacun d'eux sont égaux. Après élimination, on obtient {c, a} et {b, c}. Le premier élément du premier multi-ensemble est alors égal au deuxième élément du second multi-ensemble et on obtient {a} et {b}. La dernière comparaison établit que les deux multi-ensembles initiaux n'étaient pas égaux. Comme pour deux ensembles quelconques, l'égalité de deux multi-ensembles est définie à l'aide de l'inclusion :

Définition

*Soient E et F deux multi-ensembles. On a : (E = F) ⇔ (E ⊆ F **et** F ⊆ E)*

Autrement dit, comme pour deux ensembles quelconques, deux multi-ensembles sont égaux lorsque l'un est inclus dans l'autre et réciproquement. Soit alors **estÉgal** le prédicat qui établit l'égalité de deux tableaux considérés comme des multi-ensembles. Sa spécification est :

Algorithme 2 : Spécification de l'égalité de deux multi-ensembles

```
Algorithme estÉgal
    # t[début_t .. fin_t] et u[début_u .. fin_u] sont-ils égaux ?
Entrée
    t, u : TABLEAU[T → COMPARABLE]
    début_t, fin_t, début_u, fin_u : ENTIER
Résultat : BOOLÉEN
précondition
    index_valide(t, début_t)
    index_valide(t, fin_t)
    estDéfini(t, début_t, fin_t)
    index_valide(u, début_u)
    index_valide(u, fin_u)
    estDéfini(u, début_u, fin_u)
postcondition
    Résultat =
        (
            estInclus(t, début_t, fin_t, u, début_u, fin_u)
          et alors
            estInclus(u, début_u, fin_u, t, début_t, fin_t)
        )
```

Il reste à définir le prédicat **estInclus**. C'est une autre difficulté de cette spécification. Comme pour les ensembles, l'inclusion sera définie à l'aide de l'appartenance. Soit **appartient** le prédicat qui rend VRAI si et seulement si un élément **appartient** à un tableau.

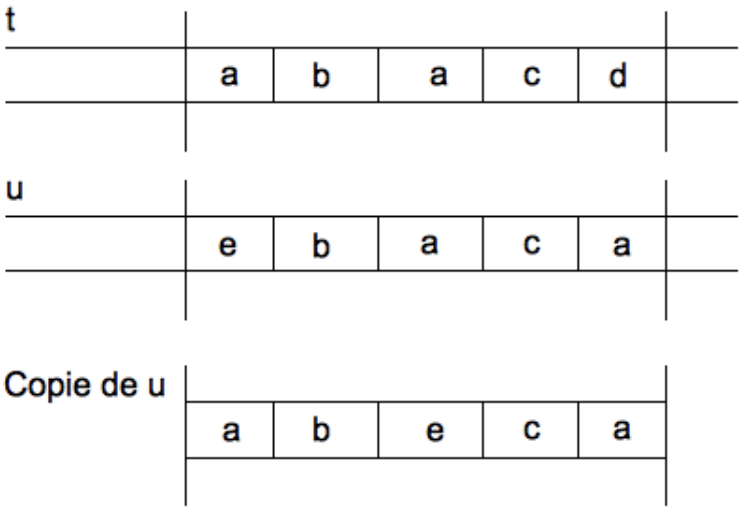
*Algorithme 3 : Spécification de **appartient** dans un tableau.*

```
Algorithme appartient
    # e appartient-il à t[début .. fin] ?
Entrée
    t : TABLEAU[T]
    début, fin : ENTIER
    e : T
Résultat : BOOLÉEN
précondition
    index_valide(t, début)
    index_valide(t, fin)
    estDéfini(t, début, fin)
postcondition
    Résultat = (rang(t, début, fin, e) ≠ ABSENT)
```

Pour s'assurer qu'un élément donné appartient au tableau, on calcule son rang, c'est-à-dire le numéro de la composante de cet élément dans le tableau. Ce rang doit être différent de ABSENT.

© ENI Editions - All rigths reserved

Pour vérifier qu'un multi-ensemble E est inclus dans le multi-ensemble F, nous devons construire un algorithme capable de vérifier l'appartenance de chacun des exemplaires d'un même élément de E à F. Pour cela, lorsqu'un élément de E a été trouvé dans F, on construit une copie de F dans laquelle l'élément commun occupe la première composante. Mais cette copie laisse F invariable. La figure ci-dessous illustre les opérations à réaliser lorsque l'élément a du tableau t a été reconnu comme un élément du tableau u.



Comme le premier élément de t et de la copie de u sont les mêmes, on peut poursuivre la même comparaison sur les mêmes ensembles, mais privés de leur premier élément. Autrement dit, il suffit de réaliser la même comparaison sur les tableaux t dans [début_t+1 .. fin_t] et copie de u dans [début_u+1 .. fin_u]. La copie de u est réalisée par la fonction **copie**, dont les spécifications sont :

*Algorithme 4 : Spécification de la fonction **copie** d'un tableau*

```
Algorithme copie
    # Réalise une copie de t[début .. fin]
Entrée
    t : TABLEAU[T]
    début, fin : ENTIER
Résultat : TABLEAU[T]
précondition
    index_valide(t, début)
    index_valide(t, fin)
    début ≤ fin
    estDéfini(t, début, fin)
postcondition
    sontIdentiques(Résultat, début, fin, t, début, fin)
fin copie
```

Le prédicat **sontIdentiques** rend VRAI si et seulement si les deux tableaux contiennent les mêmes données dans les composantes de même rang. Il sera défini précisément plus bas. Conformément à la situation présentée sur la figure précédente, on a **index_min**(**Résultat**) = début et **index_max**(**Résultat**) = fin.

C'est dans cette copie qu'est réalisé l'échange de l'élément commun à t et u. On utilise pour cela la procédure **échanger** déjà définie au chapitre Programmes directs. Soit alors **échangeInvariable** l'opération qui, partant de u, réalise la copie dans laquelle l'élément de rang k est permuté avec l'élément de rang début. Cet échange est qualifié d'invariable parce qu'il réalise l'échange sur une copie de u, donc sans modifier u. Cet échange invariable est défini par l'algorithme suivant :

Algorithme 5 : Spécification de l'échange invariable

```
Algorithme échangeInvariable
    # La copie de t dans laquelle t[début] et t[k] sont échangées.
Entrée
    t : TABLEAU[T]      # La cible de l'échange.
    début, fin : ENTIER # Numéros des composantes extrêmes à copier.
    k : ENTIER          # Numéro de la composante à échanger avec
```


© ENI Editions - All rigths reserved

```
                        # t[début].
Résultat : TABLEAU[T]
précondition
    début ≤ k ≤ fin
    index_valide(t, début)
    index_valide(t, fin)
    est_défini(t, début, fin)
postcondition
    début = fin => Résultat =
                      (début = k et Résultat[début] = t[début])
    début < fin =>
        (
            Résultat[début] = t[k]
          et
            Résultat[k] = t[début]
          et
            sontIdentiques(Résultat, début+1, k-1, t, début+1, k-1)
          et
            sontIdentiques(Résultat, k+1, fin, t, k+1, fin)
        )
fin échangeInvariable
```

Les deux dernières clauses de la postcondition expriment que les cases dont les numéros ne sont pas début et k sont recopiées sans modification. Elles s'expriment plus simplement par :

```
(∀i, début ≤ i ≤ fin et i ≠ k)(Résultat[i] = t[i])
```

Admettez que cette formule est bien plus facile à comprendre que l'expression algorithmique de l'invariance des cases correspondantes.

On peut alors donner les spécifications de **estInclus**.

*Algorithme 6 : Spécification du prédicat **estInclus***

```
Algorithme estInclus
    # t[début_t .. fin_t] ⊆ u[début_u .. fin_u] ?
Entrée
    début_t ≤ fin_t
    index_valide(t, début_t)
    index_valide(t, fin_t)
    est_défini(t, début_t, fin_t)
    début_u ≤ fin_u
    index_valide(u, début_u)
    index_valide(u, fin_u)
    est_défini(t, début_u, fin_u)
postcondition
    début_t = fin_t => Résultat =
                         appartient(u, début_u, fin_u, t[début_t])
    début_t < fin_t => Résultat =
    (
        appartient(u, début_u, fin_u, t[début_t])
      et alors
        estInclus
            (
                t, début_t, fin_t,
                échangeInvariable
                    (
                        u, début_u, fin_u,
                        rang(u, début_u, fin_u, t[début_t])
                    ),
                début_u+1, fin_u
            )
    )
fin estInclus
```

Il reste à préciser l'identité de deux tableaux. C'est fait par la définition suivante :

© ENI Editions - All rigths reserved

Définition

Deux tableaux sont identiques lorsqu'ils contiennent les mêmes éléments dans les composantes de même rang.

Le prédicat **sontIdentiques** rend VRAI si et seulement si les tableaux en paramètres sont identiques au sens de cette définition. Ses spécifications sont celles de l'algorithme ci-dessous.

*Algorithme 7 : Spécification du prédicat **sontIdentiques***

```
Algorithme sontIdentiques
    # t[début_t .. fin_t] = u[début_u .. fin_u] ?
Entrée
    t, u : TABLEAU[T]
    début_t, fin_t, début_u, fin_u : ENTIER
précondition
    début_t ≤ fin_t
    index_valide(t, début_t)
    index_valide(t, fin_t)
    est_défini(t, début_t, fin_t)
    début_u ≤ fin_u
    index_valide(u, début_u)
    index_valide(u, fin_u)
    est_défini(t, début_u, fin_u)
postcondition
    début_t = fin_t => Résultat =
        (
            début_u = fin_u
          et alors
            t[début_t] = u[début_u]
        )
    début_t < fin_t => Résultat =
        (
            t[début_t] = u[début_u]
          et alors
            sontIdentiques
                (t, début_t+1, fin_t, u, début_u+1, fin_u)
        )
fin sontIdentiques
```

L'identité de deux tableaux sur toutes leurs composantes, depuis **index_min** jusqu'à **index_max** sera notée avec le signe ' =' . Ainsi, pour deux tableaux t et u :

```
t = u ⇔ sontIdentiques
                (
                    t, index_min(t), index_max(t),
                    u, index_min(u), index_max(u)
                )
```

Il est alors possible de compléter la postcondition d'un algorithme de tri par les clauses qui précisent que les parties qui ne sont pas concernées par le tri restent identiques à leur ancienne valeur :

```
sontIdentiques
    (t, index_min(t), début-1, ancien(t), index_min(t), début-1)
sontIdentiques
    (t, fin+1, index_max(t), ancien(t), fin+1, index_max(t))
```

Ainsi, les valeurs non modifiées par le tri appartiennent à deux sous-tableaux : les composantes de numéros inférieurs à début et supérieurs à fin. Ces composantes n'existent que si **index_min**(t) < début pour le premier sous-tableau et fin < **index_max**(t) pour le second. Finalement, les clauses complètes de la postcondition qui précisent les parties non concernées par le tri sont :

```
...
index_min(t) < début =>
    sontIdentiques
```


© ENI Editions - All rigths reserved

```
        (t, index_min(t), début-1, ancien(t), index_min(t), début-1)
fin < index_max(t) =>
    sontIdentiques
        (t, fin+1, index_max(t), ancien(t), fin+1, index_max(t))
...
```

Nous disposons ainsi de tous les éléments nécessaires pour spécifier un tri. C'est fait par l'algorithme ci-dessous.

*Algorithme 8 : Spécification de **trierAsc***

```
Algorithme trierAsc
    # Trier t[début .. fin] en ordre croissant.
Entrée
    t : TABLEAU[T → COMPARABLE] # La cible du tri.
    début, fin : ENTIER # Numéros des composantes extrêmes à trier.
précondition
    index_valide(t, début)
    index_valide(t, fin)
    début ≤ fin
    estDéfini(t, début, fin)
postcondition
    # La partie triée est égale à l'ancienne.
    estÉgal(t, début, fin, ancien(t), début, fin)
    # Elle est triée en ordre croissant.
    estTriéAsc(t, début, fin)
    # Les parties non triées restent identiques.
    index_min(t) < début =>
      sontIdentiques
        (t, index_min(t), début-1, ancien(t), index_min(t), début-1)
    fin < index_max(t) =>
      sontIdentiques
        (t, fin+1, index_max(t), ancien(t), fin+1, index_max(t))
fin trierAsc
```

Étudions, à présent, quelques algorithmes de tri.

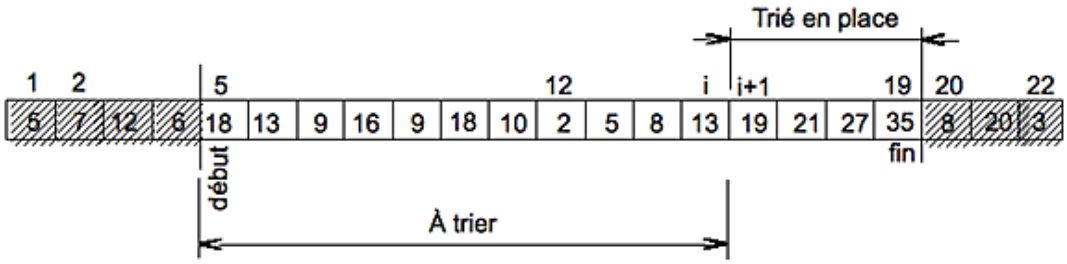
© ENI Editions - All rigths reserved

Quelques algorithmes simples

Cette section présente quelques algorithmes simples permettant de trier les éléments d'un tableau. La première partie introduit une stratégie générale naturelle de tri des éléments d'un tableau. De cette stratégie, la seconde partie décline une première solution en étudiant un algorithme qui est aux algorithmes de tri ce que le calcul d'une factorielle est à l'étude de la récursivité : un exercice obligé. Du strict point de vue informatique, les performances d'un tel algorithme sont si pauvres qu'aucun informaticien sérieux n'a songé, un seul instant, à l'utiliser en situation réelle pour trier un volume important de données. Il ne continue à être étudié que pour ses qualités pédagogiques. La troisième partie étudie une seconde solution. C'est un algorithme de tri par permutations qui effectue un peu moins d'opérations que le précédent. Ici encore, ce sont les raisonnements mis en œuvre qui justifient l'étude de cet algorithme. La dernière partie présente quelques variantes du tri par permutations.

1. Tri par permutations : introduction

Soit t un tableau de composantes de type **T** qui dérive de **COMPARABLE**. On veut le trier en ordre croissant. Cette opération consiste à amener les grandes valeurs vers le « haut » du tableau, vers la case de numéro fin. En fait, t[fin] contiendra la composante de t[début .. fin] de plus grande valeur. Une stratégie de tri consiste donc à amener la valeur maximum de t[début .. fin] en t[fin] où elle aura alors rejoint sa place définitive. La plus grande valeur de t [début .. fin-1] sera ensuite déplacée en t[fin-1] qui sera sa position définitive... Ainsi, à chaque étape du calcul, le tableau est partitionné en deux sous-tableaux : la partie « haute », qui contient les plus grands éléments et qui n'évoluera plus ; la partie « basse » qui n'est pas encore triée. Considérons, par exemple, la situation représentée par la figure ci-dessous.



La partie libellée « Trié en place » rassemble des composantes qui ont acquis leur place définitive et ne bougerons plus par la suite. Il est donc inutile d'explorer cette partie du tableau lors des prochaines phases de tri. Il suffira de faire remonter la plus grande composante de la partie « À trier » pour augmenter la partie triée et en place. Cette remarque suggère une hypothèse :

Faire une hypothèse sur l'état intermédiaire

Hypothèse (H) : La partie t[i+1 .. fin] est triée et en place.

Chaque étape du calcul ajoute une composante de plus à la partie triée et en place et, par conséquent, la valeur de `i` diminue.

Voir si c'est fini

C'est fini lorsque tout le tableau est trié et en place, donc lorsque `i + 1 = début`. Ainsi, le tri est terminé dès que `i = début - 1`, donc dès que `i ≤ début - 1`, soit dès que `i < début`. On peut réduire légèrement cet espace. Lorsque la partie triée et en place s'étend de `début + 1` à `fin`, le tableau est trié. En effet, il ne reste alors que la composante de numéro `début` et elle ne peut plus bouger puisque son déplacement déplacerait aussi l'une des autres composantes, ce qui contredirait l'hypothèse. On a donc :

`i + 1 = début + 1 ⇔ i = début`

Par conséquent, c'est fini dès que `i ≤ début`.

Se rapprocher de l'état final

On se rapproche de l'état final en ajoutant une composante de plus à la partie triée et en place. Comme cette partie ne bougera plus, la composante supplémentaire est la plus grande valeur de la partie qui n'est pas encore triée et elle prend la place de la dernière composante de cette partie. On obtient :

© ENI Editions - All rigths reserved

```
...
Amener la plus grande composante de t[début .. i] à la place de t[i]
...
```

La partie triée et en place possède alors une case de plus et s'étend, à présent, de la case de numéro `i` jusqu'à la case de numéro `fin`. Il suffit alors de reculer `i` pour retrouver **(H)**.

Initialiser le calcul

Initialement, aucune partie de `t` n'est en place. La prochaine case qui rejoindra la partie triée en place porte le numéro `i`. Initialement, c'est `fin` et, par conséquent, `i` est initialisé à `fin`.

Rédiger l'algorithme définitif

C'est fini lorsque `i = début`. Par conséquent, c'est fini lorsque `i - début = 0`. Le variant de contrôle est donc `i - début` qui est bien décroissant puisque `i` décroît de `fin` à `début + 1`.

D'autre part, dire que `t[i+1 .. fin]` est trié s'exprime par :

```
...
estTriéAsc(t, i + 1, fin)
```

Cependant, ceci n'est vrai que lorsque `i + 1 ≤ fin`, soit dès que `i < fin`. L'invariant est donc :

```
...
invariant
    i < fin => estTriéAsc(t, i + 1, fin)
...
```

Il ne faut pas seulement exprimer que cette partie du tableau est triée, mais aussi qu'elle ne bougera plus. C'est ce que montre la figure précédente, dans laquelle toutes les composantes depuis celle de numéro 12 jusqu'à celle de numéro `fin=19` sont triées en ordre croissant. Mais seule la partie des composantes de numéros 16 à `fin=19` est en place. La caractérisation précédente de l'invariant est donc insuffisante. Pour préciser que la partie depuis la case de numéro `i + 1` jusqu'à la case de numéro `fin` ne bougera plus, il suffit d'exprimer que la plus petite valeur de cette partie, celle contenue dans la case de numéro `i + 1`, est supérieure ou égale à toute valeur dans l'autre partie. Autrement dit, que cette valeur est la valeur maximum de la partie du tableau `t[début .. i+1]` :

```
...
invariant
    i < fin =>
        estTriéAsc(t, i+1, fin) et alors max(t, début, i) ≤ t[i+1]
...
```

Il reste à rédiger la solution définitive. C'est fait dans l'algorithme suivant, dans lequel la précondition et la postcondition n'ont pas été répétées.

Algorithme 9 : Tri par permutations

```
Algorithme triPermut
    # Trier t[début .. fin] en ordre croissant par permutations.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin : ENTIER
précondition
...
variable
    i : ENTIER # Numéro de la prochaine case à placer.
initialisation
    i <- fin
jusqu'à
    i ≤ début
    invariant
      i < fin =>
        estTriéAsc(t, i+1, fin) et alors max(t, début, i) ≤ t[i+1]
    variant de contrôle
        i - début
```


© ENI Editions - All rigths reserved

```
répéter
    Amener la plus grande composante de t[début .. i] en t[i]
    i <- i - 1
fin répéter
postcondition
...
fin triPermut
```

Il reste à donner un sens à l'expression « Amener la plus grande composante de t[début .. i] en t[i] ». On obtient différentes solutions selon sa réalisation. La section suivante en étudie une qui vient naturellement à l'esprit.

2. Tri par permutations

Il s'agit donc d'amener la composante de t[début .. i] de valeur maximum en t[i]. Nous savons déterminer la position de la composante maximum d'un tableau. Il s'agit d'utiliser la fonction **rangDuMax** qui n'a pas encore été définie. Un chapitre antérieur a étudié la fonction duale **rangDuMin** et la nouvelle fonction se définit de la même façon. Cette définition est laissée en exercice. On obtient donc :

```
...
répéter
    # Amener la plus grande valeur de t[début .. i] à la place
    # de t[i].
    k <- rangDuMax(t, début, i)
    échanger(t[i], t[k])
    assertion
        i < fin =>
            estTriéAsc(t, i, fin)
          et alors
            max(t, début, i - 1) ≤ t[i+1]
    i <- i - 1
fin répéter
...
```

Ce segment d'algorithme fait apparaître un paragraphe **assertion** qui décrit l'état intermédiaire atteint en utilisant les prédicats habituels. Il termine la première solution au problème posé. Il reste à rassembler les différentes parties étudiées pour obtenir un algorithme complet.

Exercice 1 : Extensions au tri par permutations

Cet exercice propose quelques variations sur le thème précédent.

1. Au lieu d'un tri en ordre croissant, définir un tri en ordre décroissant.

2. Faire l'hypothèse que c'est la partie t[début .. i - 1] qui est triée en place et redéfinir le tri en ordre croissant.

3. Pour le tri de la question précédente, écrire avant et après chaque instruction de l'itération l'assertion algorithmique qui décrit l'état intermédiaire obtenu. Définir les prédicats nécessaires et mettre en évidence la conservation ou non de l'invariant.

La section suivante étudie une autre solution, obtenue en dérivant d'une autre façon la partie de l'algorithme qui fait remonter la composante maximum vers sa place définitive.

3. Tri « à bulle » (Bubble Sort)

Le tri de la section précédente fait beaucoup de travail inutile. Ainsi, quand le maximum d'une partie du tableau est déterminé, l'itération suivante recommence pour déterminer le maximum de la même partie, mais privée d'un élément. On décide d'exploiter une exploration du tableau en profitant de la rencontre de deux éléments, qui ne sont pas en ordre, pour les permuter. Pour réaliser cette exploration, on procède ainsi : on compare chaque élément à tous ceux qui le précèdent. Une permutation est effectuée chaque fois que deux éléments ne sont pas en ordre. Ainsi, l'élément courant, celui de rang i + 1, est successivement comparé aux éléments de rangs j = i, j = i – 1… jusqu'à celui de rang début. Cette description suggère donc une hypothèse :

Faire une hypothèse sur l'état actuel

Hypothèse (H) : t[j+1 .. i] a été exploré et les éléments qui ne sont pas en ordre ont été permutés.

© ENI Editions - All rigths reserved

Voir si c'est fini

C'est fini lorsque j + 1 = début, donc dès que j = début – 1, soit dès que j < début. L'invariant peut être immédiatement précisé. Comme la permutation des éléments qui n'étaient pas en ordre a été effectuée, on a :

```
...
invariant
    t[i] = max(t, j + 1, i)
...
```

De même, le variant de contrôle se déduit immédiatement de la condition de terminaison :

```
...
variant de contrôle
    début - j - 1
...
```

Se rapprocher de l'état final

On se rapproche de l'état final en comparant une case de plus à la case courante, celle de numéro i . La case à comparer est celle de numéro j , d'où :

```
...
si
    t[j] > t[i]
alors
    échanger(t[j], t[i])
fin si
j <- j - 1
...
```

Initialiser le calcul

Pour réaliser l'hypothèse, il convient d'initialiser j de façon que t[j+1 .. i] ait déjà été traité. La prochaine case à observer est celle de numéro j . Initialement, aucune case n'a encore été observée et donc, la première case à observer est celle de numéro i . On a donc :

```
...
initialisation
    j <- i
...
```

Rédiger l'algorithme définitif

D'où la réalisation définitive :

```
...
# Amener la plus grande valeur de t[début .. i] à la place de t[i].
initialisation
    j <- i
jusqu'à
    j < début
    invariant
        j < i => t[i] = max(t, j + 1, i)
    variant de contrôle
        début - j - 1
répéter
    si
        t[j] > t[i]
    alors
        échanger(t[i], t[j])
    fin si
    j <- j - 1
```


© ENI Editions - All rigths reserved

```
fin répéter
...
```

Il ne reste plus qu'à rassembler les différentes parties pour obtenir l'algorithme définitif et complet appelé « tri à bulle » (*bubble sort*). Cet algorithme tient son nom de ce que, à chaque exploration, le plus grand élément qui n'est pas encore à sa place remonte vers sa position définitive, comme une bulle remonte à la surface d'un récipient rempli de liquide.

4. D'autres tris par permutations

À chaque fois que l'on rencontre deux composantes qui se suivent et qui ne sont pas en ordre, on les échange. On recommence une exploration complète du tableau à chaque fois que la dernière exploration a procédé à au moins un échange. Le tableau est entièrement trié lorsqu'une exploration complète n'a provoqué aucun échange. La table ci-dessous montre un exemple de l'évolution d'un tableau. La première colonne est le numéro d'ordre de l'exploration. Chaque ligne est une exploration du tableau à trier.

t	5	3	8	12	6	4
1	3	5	8	6	4	12
2	3	5	6	4	8	12
3	3	5	4	6	8	12
4	3	4	5	6	8	12
t	3	4	5	6	8	12

La première exploration rencontre successivement 5 et 3 puis 12, 6 et 4 qui ne sont pas en ordre. 5 et 3, 12 et 6 puis 12 et 4 sont permutés et la ligne 1 du tableau est obtenue. Dans cette ligne, 12 est remonté de sa position initiale à sa position définitive. L'exploration de la ligne 2 rencontre, dans la ligne 1, les valeurs 8 et 6 qui ne sont pas en ordre et les permute. Elle trouve alors 8 et 4 en désordre et les permute. Ainsi, 8 est remonté de sa position initiale, en ligne 1, à sa position définitive, en ligne 2. Les valeurs 8 et 12 sont alors dans leur position définitive et une nouvelle exploration est entreprise. En itérant ce procédé, on arrive à la ligne 4. Une nouvelle exploration ne provoque plus aucune permutation d'élément et l'algorithme se termine.

L'algorithme, pour trier un tableau selon cette stratégie, consiste donc en une succession d'explorations complètes du tableau. Soit **triAperm** cet algorithme ; son écriture est simple et ne mérite pas plus de développement dans sa forme naïve. Une variable booléenne `trié` prend la valeur FAUX, si au moins une permutation de valeurs dans `t` a eu lieu lors de la dernière exploration. Chaque exploration commence à la case de numéro début et compare toutes les paires de composantes jusqu'à celle de numéro `fin`. On obtient l'algorithme suivant, dans lequel la précondition et la postcondition ne sont pas répétées.

Algorithme 10 : Algorithme du tri par permutations : version provisoire

```
Algorithme triAperm
    # Tri de t[début .. fin] en ordre croissant par permutations.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin : ENTIER
précondition
...
variable
    trié : BOOLÉEN
initialisation
    trié <- FAUX
répéter
    explorer(t, début, fin, trié)
jusqu'à trié
fin répéter
postcondition
...
fin triAperm
```

© ENI Editions - All rigths reserved

Cette expression traduit directement la description informelle du comportement de l'algorithme. Remarquer la construction **répéter ... jusqu'à**. Elle indique une itération, mais dans laquelle la condition de terminaison est évaluée juste avant la clause habituelle **fin répéter**. C'est que l'exploration doit être réalisée au moins une fois quelle que soit la valeur initiale de la variable `trié` qui contrôle l'itération. D'ailleurs, l'initialisation de cette variable est inutile puisque c'est la responsabilité de l'exploration que de lui donner une valeur. On peut donc supprimer cette initialisation sans conséquence. La version définitive est celle de l'algorithme qui suit.

Algorithme 11 : Algorithme du tri par permutations - version naïve définitive

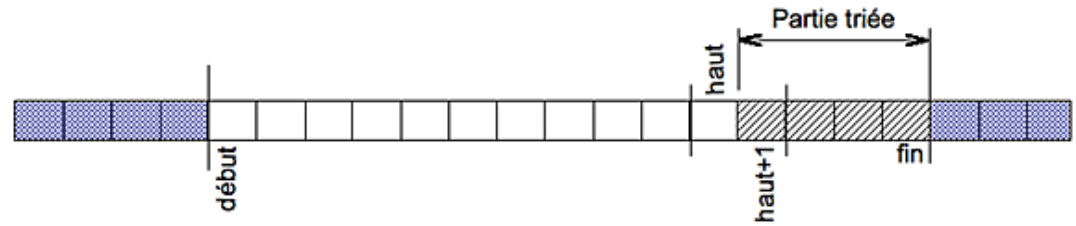
```
Algorithme triAperm
    # Tri de t[début .. fin] en ordre croissant par permutations.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin : ENTIER
précondition
...
variable
    trié : BOOLÉEN
répéter
    explorer(t, début, fin, trié)
jusqu'à trié
fin répéter
postcondition
...
fin triAperm
```

Bien entendu, moyennant quelques adaptations, il est possible de continuer à utiliser la construction **jusqu'à ... répéter** ou **tant que ... répéter** pour définir cet algorithme. On voit ainsi qu'il existe différentes façons d'exprimer sans ambiguïté les instructions d'un algorithme.

Cette version est facile à comprendre, simple à spécifier et à écrire, mais elle fait trop de travail inutile. Chaque exploration vérifie l'ordre des composantes jusqu'à la dernière. Or, nous avons vu dans l'exemple de la table précédente que, à chaque passage, au moins la valeur maximum de la partie non triée rejoint sa position définitive. Il est donc inutile de terminer un passage à la composante de numéro `fin` : on peut s'arrêter avant. L'hypothèse est donc encore qu'une partie du tableau a rejoint sa position définitive.

Faire une hypothèse sur l'état actuel

Dans un état intermédiaire, on a déjà réalisé des explorations du tableau et on a procédé à un certain nombre de permutations. Elles ont amené des éléments à leur place, en haut du tableau. La figure ci-dessous représente une telle situation.



Cette figure représente une situation dans laquelle les permutations précédentes ont amené les composantes les plus grandes dans `t[haut+1 .. fin]`. Les composantes en bas du tableau, depuis celle de numéro début jusqu'à celle de numéro haut, si elles existent, doivent encore être triées. D'où, l'hypothèse décrivant cette situation :

Hypothèse (H) : des permutations ont amené la partie t[haut+1 .. fin] à sa place.

Dire que cette partie est à sa place, c'est dire d'abord qu'elle est triée :

```
...
invariant
    haut < fin => estTrieAsc(t, haut + 1, fin)
...
```

C'est dire ensuite qu'elle « ne bougera plus ». Autrement dit, les éléments de cette partie ont acquis leur position définitive. Or, un élément est déplacé lorsque l'on trouve, dans le tableau, un élément plus grand que lui. Par conséquent, dire qu'un élément ne bougera plus, c'est dire que tout élément qui n'est pas encore à sa place lui est

© ENI Editions - All rigths reserved

inférieur ou égal. Ainsi, la plus grande valeur des éléments qui n'ont pas encore rejoint leur place reste inférieure ou égale à tout élément qui a rejoint sa place définitive. Par conséquent, la valeur maximum, dans le tableau non trié, est inférieure ou égale à la valeur la plus petite des éléments à leur place. Ceci s'écrit :

```
...
invariant
    ...
    et max(t, début, haut) ≤ t[haut+1]
...
```

Voir si c'est fini

C'est fini dès que la partie triée recouvre le tableau, donc dès que haut = début. Le variant de contrôle est donc la différence haut – début.

Se rapprocher de l'état final

On se rapproche de l'état final en réalisant une exploration de plus de la partie non triée. Au cours de cette exploration, on permute les valeurs consécutives rencontrées lorsqu'elles ne sont pas en ordre. Cette exploration adresse le tableau depuis la case de numéro début jusqu'à la case de numéro haut. Elle met trié à FAUX lorsque se produit une permutation, signe qu'une valeur remonte vers sa position définitive. On se rapproche de l'état final en appelant les services de la procédure **explorer** entre ces deux valeurs :

```
...
répéter
    explorer(t, début, haut, trié)
    # t[haut .. fin] est à sa place.
...
```

On retrouve **(H)** en reculant haut :

```
...
répéter
    explorer(t, début, haut, trié)
    # t[haut .. fin] est à sa place.
    haut <- haut - 1
    # t[haut - 1 .. fin] est à sa place.
fin répéter
...
```

Il reste à initialiser le calcul.

Initialiser le calcul

Initialement, le tableau n'est pas trié et la première exploration observe toutes les cases du tableau. Par conséquent, haut = fin. On a :

```
...
initialisation
    haut <- fin
    trié <- FAUX
...
```

Rédiger l'algorithme définitif

Il ne reste plus qu'à rédiger l'algorithme définitif, ce qui est une formalité avec ce qui a été dit plus haut. On obtient :

Algorithme 12 : Algorithme du tri par permutations - Version définitive

```
Algorithme triAperm
    # Trier t[début .. fin] en ordre croissant.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin : ENTIER
précondition
```

© ENI Editions - All rigths reserved

```
...
variable
    trié : BOOLÉEN
    haut : ENTIER
initialisation
    trié <- FAUX
    haut <- fin
invariant
    haut < fin =>
        (
            estTrieAsc(t, haut + 1, fin)
          et
            max(t, début, haut) ≤ t[haut+1]
        )
variant de contrôle
    haut - début
répéter
    explorer(t, debut, haut, trié)
    # t[haut .. fin] est à sa place.
    haut <- haut - 1
    # t[haut - 1 .. fin] est à sa place.
jusqu'à trié
postcondition
...
fin triAperm
```

Il reste à spécifier et à réaliser la procédure **explorer**. La spécification reprend les clauses d'un tri qui précisent que le tableau t n'est pas modifié dans les intervalles externes à [début .. fin] et que les mêmes composantes restent dans cet intervalle, peut-être après avoir changé de place. On a donc :

```
postcondition
    # La partie explorée est égale à l'ancienne.
    estÉgal(t, début, fin, ancien(t), début, fin)
    # Les parties non explorées restent identiques.
    index_min(t) < début =>
        sontIdentiques
                (
                t, index_min(t), début-1,
                ancien(t), index_min(t), début-1
                )
    fin < index_max(t) =>
        sontIdentiques
                (
                t, fin+1,index_max(t),
                ancien(t), fin+1, index_max(t)
                )
...
```

Mais il faut encore dire ce que fait subir cette procédure à la partie explorée. C'est une spécification difficile à écrire. On se contente ici de l'exprimer en français :

```
...
    # Permutations sur la partie explorée.
    Chaque composante de t[début .. fin] est placée après la plus
    grande suite de composantes dont elle est le max.
...
```

Soit **estExploré** le prédicat qui exprime cette assertion. Sa spécification est la suivante :

*Algorithme 13 : Spécification de **estExploré***

```
Algorithme estExploré
    # VRAI ssi une exploration complète de t[début .. fin} a été
    # réalisée.
Entrée
    t : TABLEAU[T → COMPARABLE]
```


© ENI Editions - All rigths reserved

```
    début, fin : ENTIER
Résultat : BOOLÉEN
précondition
    début ≤ fin
    estDéfini(t)
    index_valide(t, début)
    index_valide(t, fin)
postcondition
    # La partie explorée est égale à l'ancienne.
    estÉgal(t, début, fin, ancien(t), début, fin)
    # Permutations sur la partie explorée.
    Chaque composante de t[début .. fin] est placée après la plus
    grande suite de composantes dont elle est le max.
    # Les parties non explorées restent identiques.
    index_min(t) < début =>
        sontIdentiques
                (
                t, index_min(t), début-1,
                ancien(t), index_min(t), début-1
                )
    fin < index_max(t) =>
        sontIdentiques
                (
                t, fin+1,index_max(t),
                ancien(t), fin+1, index_max(t)
                )
fin estExploré
```

On peut alors étudier la réalisation de l'algorithme. Il parcourt le tableau et, pour chaque composante strictement supérieure à celle qui suit, il réalise un échange ; dans un état intermédiaire, on a déjà réalisé une partie du parcours :

Faire une hypothèse sur l'état intermédiaire

Hypothèse (H) : t a été exploré jusqu'à la composante de numéro (i - 1). 'trié' est VRAI s'il n'y a pas eu d'échange réalisé.

L'expression « t a été exploré » signifie que, lors de cette exploration, les composantes strictement supérieures à celles qui les suivent ont été permutées.

Voir si c'est fini

C'est fini lorsque tout le tableau a été exploré entre début et fin. Comme l'exploration est réalisée jusqu'à la composante de numéro (i – 1) d'après **(H)**, c'est que cette dernière a été comparée à celle qui la suit, c'est-à-dire à la composante de numéro i. C'est fini lorsque i = fin :

```
...
    si
        i = fin
    alors
        c'est fini
    sinon
...
```

Sinon, il faut comparer la composante de numéro i à celle qui la suit, de numéro (i + 1).

Se rapprocher de l'état final

```
...
    sinon
        si
            t[i] > t[i+1]
        alors
            échanger(t[i], t[i+1])
            trié <- FAUX
```

© ENI Editions - All rigths reserved

```
        fin si
...
```

Le système est alors dans une nouvelle situation dans laquelle une case de plus a été explorée. Ce nouvel état est décrit par :

```
...
        assertion
            t a été exploré jusqu'à la composante de numéro i
...
```

On retrouve l'état décrit par **(H)** en avançant i :

```
...
        assertion
            t a été exploré jusqu'à la composante de numéro i
        i <- i + 1
        assertion
            t a été exploré jusqu'à la composante de numéro i - 1
...
```

et alors, les mêmes calculs peuvent être itérés.

Initialiser le calcul

Il faut rendre l'énoncé de **(H)** vrai dans l'état initial. La prochaine case à explorer est celle de numéro i. Au commencement, c'est la case de numéro début. Par conséquent, on obtient :

```
...
initialisation
    i <- début
...
```

De plus, on a que la variable trié a la valeur VRAI si aucun échange n'a été réalisé d'après **(H)**. Initialement, aucun échange n'a encore été réalisé et donc :

```
...
initialisation
    i <- début
    trié <- VRAI
...
```

Rédiger l'algorithme définitif

L'hypothèse **(H)** exprime que le tableau a été exploré jusqu'à la composante de numéro (i – 1) et c'est exactement ce que dit **estExploré** lorsque fin = (i – 1). D'où l'invariant :

```
...
invariant
    i > début => estExploré(t, début, i - 1)
...
```

C'est fini dès que i = fin, donc dès que fin – i = 0. Comme i est croissant de début à fin, c'est fini dès que i ≥ fin. Ainsi, on a :

```
...
jusqu'à
    i ≥ fin
variant de contrôle
    fin - i
...
```

L'algorithme définitif est donné ci-dessous.

*Algorithme 14 : Algorithme de la procédure **explorer** - Version 1.0*

© ENI Editions - All rigths reserved

```
Algorithme explorer
    # Faire remonter chaque composante de t[début .. fin] à la fin
    # de la plus grande suite dont elle est le max.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin : ENTIER
    trié : BOOLÉEN
précondition
    début ≤ fin
    estDéfini(t)
    index_valide(t, début)
    index_valide(t, fin)
variable
    i : ENTIER # Numéro prochaine case à explorer.
initialisation
    i <- début
    trié <- VRAI
jusqu'à
    i ≥ fin
    invariant
        i > début => estExploré(i, début, i - 1)
    variant de contrôle
        fin - i
répéter
    si
        t[i] > t[i+1]
    alors
        échanger(t[i], t[i+1])
        trié <- FAUX
    fin si
    i <- i + 1
fin répéter
postcondition
    estExploré(t, début, fin)
fin explorer
```

Remarquez que, dans cet algorithme de l'exploration, chaque élément est comparé à celui qui le suit immédiatement. C'est ce que réalisent les instructions :

```
...
    si
        t[i] > t[i+1]
    alors
        échanger(t[i], t[i+1])
...
```

On peut qualifier cette exploration en disant qu'elle explore le tableau avec un « pas » d'une unité, c'est-à-dire d'une case. Une idée difficile à justifier mathématiquement consiste à explorer le tableau avec un pas supérieur à 1, mais inférieur à la demi-taille du tableau, c'est-à-dire à la moitié du nombre de composantes. L'algorithme de l'exploration devient celui qui suit, dans lequel un nouveau paramètre, pas, précise avec quel pas doit être réalisée l'exploration.

*Algorithme 15 : Algorithme de la procédure **explorer** - Version 2.0*

```
Algorithme explorer
    # Faire remonter chaque composante de t[début .. fin] à la fin
    # de la plus grande suite dont elle est le max.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin, pas : ENTIER
    trié : BOOLÉEN
précondition
    début ≤ fin
    1 ≤ pas ≤ quotient(fin - début + 1, 2)
    estDéfini(t)
    index_valide(t, début)
```

© ENI Editions - All rigths reserved

```
    index_valide(t, fin)
variable
    i : ENTIER
initialisation
    i <- début
    trié <- VRAI
jusqu'à
    i > fin - pas
    invariant
        i > début => estExploré(i, début, i - 1, pas)
    variant de contrôle
        fin - pas - i + 1
répéter
    si
        t[i] > t[i+pas]
    alors
        échanger(t[i], t[i+pas])
        trié <- FAUX
    fin si
    i <- i + 1
fin répéter
postcondition
    estExploré(t, début, fin)
fin explorer
```

Cette procédure est utilisée comme précédemment, jusqu'à ce que l'exploration ne permute plus aucun élément. Une nouvelle campagne d'explorations est alors entreprise avec un pas réduit de moitié. Le tableau est alors trié lorsque le pas devient nul. On obtient l'algorithme suivant :

Algorithme 16 : Tri par permutations avec un pas variable

```
Algorithme trier
    # Trier t[début .. fin] en ordre croissant.
Entrée
    t : TABLEAU[T → COMPARABLE]
    début, fin, p : ENTIER
précondition
...
variable
    trié : BOOLÉEN
    pas  : ENTIER
initialisation
    pas <- p
jusqu'à
    pas < 1
répéter
    répéter
        explorer(t, début, fin, pas, trié)
    jusqu'à
        trié
    fin répéter
    pas <- quotient(pas, 2)
fin répéter
postcondition
...
fin trier
```

L'exploration de l'algorithme 14 est obtenue en utilisant ce dernier algorithme avec un pas de 1 :

```
...
# tri par permutations simple.
trier(t, début, fin, 1)
...
```

Mais on peut démontrer que les performances sont meilleures, en ce sens que le nombre de comparaisons/permutations est moindre, lorsque le pas initial est égal à la demi-taille du tableau à trier :

© ENI Editions - All rigths reserved

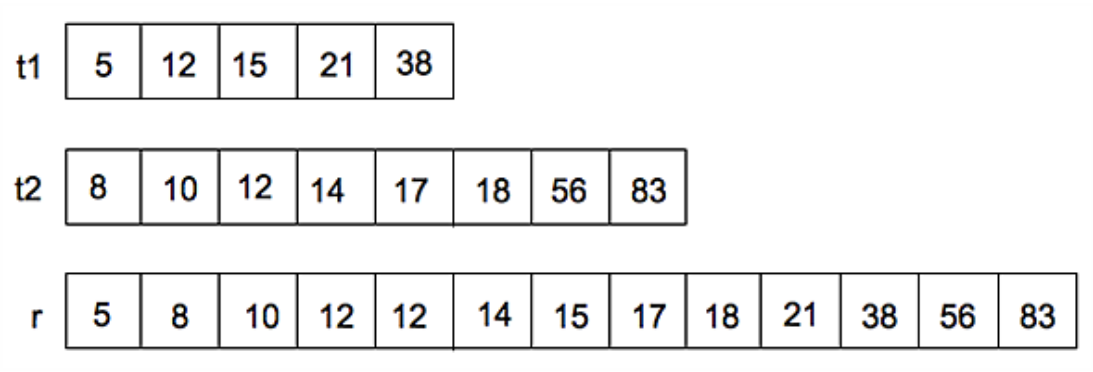
```
...
# tri par permutations « shell ».
variable
    taille, pas : ENTIER
initialisation
    taille <- fin - début + 1
    pas <- quotient(taille, 2)

trier(t, début, fin, pas)
...
```

Cet algorithme est appelé le « tri de *shell* ».

© ENI Editions - All rigths reserved

Fusionner deux tableaux triés

On donne deux tableaux, t1 et t2 triés en ordre croissant. On veut obtenir un tableau **Résultat** de même type, trié en ordre croissant, qui regroupe les éléments de t1 et t2. La figure ci-dessous illustre l'effet de l'algorithme à étudier.



Remarquez que les deux tableaux n'ont pas nécessairement le même nombre d'éléments. Le tableau résultat aura un nombre de composantes égal à la somme des nombres des composantes de `t1` et `t2`. Les éléments identiques dans les deux tableaux sont répétés dans le tableau résultat. D'ailleurs, un même tableau peut contenir des éléments en plusieurs exemplaires.

Si on fusionne `t1` entre les cases `début1` et `fin1` avec `t2` entre les cases `début2` et `fin2`, on obtient un tableau résultat qui aura `fin1 - début1 + 1 + fin2 - début2 + 1` composantes. Afin de résoudre ce problème facilement, la section suivante commence par introduire un nouveau type de données : le *vecteur*. C'est un tableau sur lequel sont définies des opérations applicables particulières. La section qui la suivra permettra alors de résoudre le problème de la fusion de deux vecteurs.

1. Définition d'un vecteur

Un *vecteur* est un tableau sur lequel est défini un *curseur*. C'est un numéro qui repère une case particulière qui sera la case courante du tableau, c'est-à-dire la case sur laquelle opèrent implicitement certaines des opérations applicables à un vecteur. Des opérations particulières permettent de déplacer le curseur ou d'accéder à la composante du tableau située « sous le curseur », c'est-à-dire repérée par le curseur. Un vecteur est défini par :

```
type
    VECTEUR
structure
    t : TABLEAU[T]   # Données associées au vecteur.
    début   : ENTIER# Première case du vecteur dans t.
    fin     : ENTIER # Dernière case du vecteur dans t.
    curseur : ENTIER # Numéro de la case courante.
invariant
    début ≤ fin
    début - 1 ≤ curseur ≤ fin + 1
    index_valide(t, début)
    index_valide(t, fin)
fin VECTEUR
```

Un vecteur est déclaré comme un tableau, en précisant le type **T** des données qu'il contient et les numéros des cases extrêmes, comme ceci :

```
...
variable
    v : VECTEUR[ENTIER][-15, 30]
...
```

Cette « instruction » déclare un vecteur v d'entiers, dont les cases seront numérotées de -15 à 30. Pour utiliser un vecteur, il doit évidemment avoir été initialisé. Cette initialisation consiste à donner une valeur à `début` et à `fin`, à remplir le tableau `v.t` de données, au moins entre `début` et `fin` et, enfin, à positionner le curseur sur la première case utile. Cette initialisation étant réalisée, le prédicat **estDéfini** appliqué au vecteur retourne VRAI et les opérations définies dans les paragraphes qui suivent deviennent utilisables.

Les opérations applicables à un vecteur font toutes intervenir le curseur. Elles se répartissent en trois classes :

© ENI Editions - All rigths reserved

- les opérations qui permettent de connaître la position actuelle du curseur, sauf la dernière :
    - **estAvant** (respectivement **estAprès**) est un prédicat qui rend VRAI si et seulement si le curseur est placé avant la première case (respectivement après la dernière case) ;
    - **estPremier** (respectivement **estDernier**) rend VRAI si et seulement si le curseur est placé sous la première case (respectivement sous la dernière case) ;
    - **position** est une fonction qui rend le numéro de la case repérée par le curseur ;
- les opérations qui permettent d'accéder à la donnée contenue dans la case repérée par le curseur :
    - **lire** rend une copie de la donnée dans la case sous le curseur ;
    - **écrire** est une procédure qui place une donnée précisée en paramètre dans la case repérée par le curseur. La donnée contenue dans cette case avant l'écriture est perdue ;
- les opérations permettant de déplacer le curseur :
    - **premier** (respectivement **dernier**) place le curseur sous la première case (respectivement sous la dernière case) ;
    - **avant** (respectivement **après**) place le curseur avant la première case (respectivement après la dernière case) ;
    - **suivant** (respectivement **précédent**) avance (respectivement recule) la position du curseur d'une case ;
    - **avancer** (respectivement **reculer**) permet d'avancer (respectivement de reculer) le curseur d'un nombre de positions précisé en paramètre ;
    - **allerA** place le curseur sous une case dont le numéro est précisé en paramètre ;
- La requête **cardinal** rend le nombre de composantes du vecteur.

Il s'agit, à présent, de définir ces opérations. Commençons par les utiliser pour résoudre un problème simple afin de montrer comment les mettre en œuvre.

Exercice résolu 2 : Calcul de la moyenne arithmétique des composantes d'un vecteur

On donne un vecteur v de nombres réels.

1. Faire un algorithme qui calcule la moyenne arithmétique des composantes de v.

Solution

L'algorithme **moyenne** prend en entrée un vecteur défini de nombres réels et retourne la moyenne arithmétique de ses composantes. Les spécifications sont données par l'algorithme ci-dessous qui traduit directement la définition d'une moyenne.

Algorithme 17 : Définition de la moyenne arithmétique d'un vecteur de réels

```
Algorithme moyenne
    # La moyenne arithmétique des composantes de v.
Entrée
    v : VECTEUR[RÉEL]
Résultat : RÉEL
précondition
    estDéfini(v)
variable
```


© ENI Editions - All rigths reserved

```
    n : ENTIER
initialisation
    n <- cardinal(v)
réalisation
    si
        n = 0
    alors
        Résultat <- INFINI
    sinon
        Résultat <- somme(v) / n
    fin si
postcondition
    cardinal(v) = 0 => Résultat = INFINI
    cardinal(v) > 0 => Résultat = somme(v) / n
fin moyenne
```

L'algorithme calcule d'abord le nombre d'éléments du vecteur et n'entreprend le calcul de la somme des composantes que si ce nombre n'est pas nul. La somme des composantes est déterminée par une fonction **somme** qui reste à définir. Remarquez que, en toute rigueur, il n'est pas nécessaire de se préoccuper de la valeur de n. En effet, l'invariant d'un vecteur défini précise que `début ≤ fin`. Par conséquent, le vecteur n'est défini que s'il possède au moins une composante utile et donc **cardinal**(v) > 0. Dans le cas contraire, la précondition est fausse. Cette remarque permet donc de simplifier la spécification qui précède. Cette simplification est laissée en exercice.

L'algorithme suivant définit une version incorrecte de la fonction **somme**.

*Algorithme 18 : Version incorrecte de la fonction **somme***

```
Algorithme somme
    # La somme des composantes de v.
Entrée
    v : VECTEUR[RÉEL]
Résultat : RÉEL
précondition
    estDefini(v)
initialisation
    Résultat <- 0
    avant(v) # Place le curseur avant la première composante.
Jusqu'à estAprès(v)
répéter
    Résultat <- Résultat + lire(v)
    suivant(v)
fin répéter
postcondition
Résultat =
```

$$\frac{\sum_{i=v.début}^{i=v.fin} v.t[i]}{cardinal(v)}$$

```
 fin somme
```

Cette solution n'est pas correcte. L'algorithme parcours le vecteur pour déterminer la valeur demandée. *Le parcours modifie la valeur du curseur et, par conséquent, le vecteur lui-même* puisque la valeur du curseur est un champ de la structure qui le définit. Pour obtenir une fonction dans ces conditions, nous devons réaliser une copie du vecteur sur laquelle elle opère, afin de ne pas introduire d'effet de bord. On obtient alors l'algorithme suivant qui utilise une fonction **copie** pour réaliser une copie de travail du vecteur et ainsi le laisser invariable.

*Algorithme 19 : Version définitive de la fonction **somme***

```
Algorithme somme
    # La somme des composantes de v.
Entrée
    v : VECTEUR[RÉEL]
```

© ENI Editions - All rigths reserved

```
Résultat : RÉEL
précondition
    estDefini(v)
variable
    cv : VECTEUR[RÉEL] # Copie de v.
initialisation
    cv <- copie(v, v.début, v.fin)
    Résultat <- 0
    avant(cv) # Place le curseur avant la première composante.
Jusqu'à estAprès(cv)
répéter
    Résultat <- Résultat + lire(cv)
    suivant(cv)
fin répéter
postcondition
Résultat = 
fin somme
```

$$\frac{\sum_{i=v.début}^{i=v.fin} v.t[i]}{cardinal(v)}$$

Il reste à définir l'opération **cardinal** qui rend le nombre de composantes d'un vecteur. Sa définition est simple et elle est laissée en exercice.

L'exercice suivant propose l'étude des opérations applicables à un vecteur et leur mise en œuvre pour définir une fonction.

Exercice résolu 2 : Définition des opérations applicables à un vecteur

Cet exercice demande de préparer la définition de chacune de ces opérations.

1. Donner les spécifications de chacune des opérations applicables à un vecteur.

2. Utiliser ces opérations pour refaire l'algorithme qui détermine la valeur de la composante minimum d'un vecteur.

3. Donner alors la réalisation de chaque opération.

Solution

Définition des prédicats et de la fonction **position**.

Le prédicat **estAvant** rend VRAI si le curseur est placé avant la première case du vecteur. C'est le cas lorsque `curseur ≤ début - 1`. La définition du prédicat est donc :

*Algorithme 20 : Spécification du prédicat **estAvant***

```
Algorithme estAvant
    # Le curseur est-il avant la première case du vecteur ?
Entrée
    v : VECTEUR[T]
Résultat : BOOLÉEN
précondition
    estDéfini(v)
postcondition
    Résultat = (v.curseur < début)
fin estAvant
```

On suppose que ce prédicat rend VRAI dès que le vecteur est défini et avant toute première utilisation.

estAprès est le prédicat dual du précédent et il se définit de la même façon. Ses spécifications sont laissées en exercice. De même, **estPremier** rend VRAI si et seulement si le curseur est sous la première case, c'est-à-dire si et seulement si `v.curseur = v.début`.

Le prédicat **estDernier** s'écrit de la même façon. Ses spécifications sont laissées en exercice.

La fonction **position** ne fait que renvoyer le numéro de la case repérée par le curseur. Par conséquent, elle ne fait que renvoyer la valeur du curseur :

© ENI Editions - All rigths reserved

*Algorithme 21 : Spécification de la fonction **position***

```
Algorithme position
    # Valeur actuelle du curseur i.e numéro de la case qu'il repère.
Entrée
    v : VECTEUR[T]
Résultat : ENTIER
précondition
    estDéfini(v)
postcondition
    Résultat = v.curseur
fin position
```

Procédures de déplacement du curseur

La procédure **premier** positionne le curseur sous la première case du tableau :

*Algorithme 22 : Spécification de la procédure **premier***

```
Algorithme premier
    # Place le curseur sous la première case de v.
Entrée
    v : VECTEUR[T]
précondition
    estDéfini(v)
postcondition
    estPremier(v)
    v.t = ancien(v).t
    v.début = ancien(v.début)
    v.fin   = ancien(v.fin)
fin premier
```

La procédure **dernier** se définit de la même façon. Elle est laissée en exercice.

La procédure **avant** place le curseur avant la première case utile. On a donc :

*Algorithme 23 : Spécification de la procédure **avant***

```
Algorithme avant
    # Place le curseur avant la première case de v.
Entrée
    v : VECTEUR[T]
précondition
    estDéfini(v)
postcondition
    estAvant(v)
    v.t     = ancien(v).t
    v.début = ancien(v.début)
    v.fin   = ancien(v.fin)
fin avant
```

La procédure **après** est définie de la même façon. Les procédures **suivant** et **précédent** ne sont pas plus difficiles à spécifier :

*Algorithme 24 : Spécification de la procédure **suivant***

```
Algorithme suivant
    # Avance le curseur d'une case.
Entrée
    v : VECTEUR[T]
précondition
    estDéfini(v)
postcondition
```

© ENI Editions - All rigths reserved

```
    v.t     = ancien(v).t
    v.début = ancien(v).début
    v.fin   = ancien(v).fin
    estAprès(ancien(v))      => v = ancien(v)
    non estAprès(ancien(v)) => position(v) = position(ancien(v)) + 1
fin premier
```

On ne peut donc avancer le curseur d'une case lorsqu'il a déjà dépassé la dernière. De même, on ne peut le reculer quand il est déjà avant la première.

La procédure **avancer** permet d'avancer le curseur d'un nombre de cases précisé. L'algorithme suivant en donne la spécification.

*Algorithme 25 : Spécification de **avancer***

```
Algorithme avancer
    # Avance le curseur de p positions.
Entrée
    v : VECTEUR[T]
    p : ENTIER
précondition
    estDéfini(v)
    position(v) + p < v.fin + 1
postcondition
    position(v) = position(ancien(v)) + p
    v.t     = ancien(v).t
    v.début = ancien(v).début
    v.fin   = ancien(v).fin
fin avancer
```

On spécifie de même la procédure duale **reculer**. Remarquez aussi que, à quelques adaptations près de la précondition, **reculer** de `p` c'est **avancer** de `-p`. Cette remarque permet de factoriser une partie des instructions de ces deux procédures.

Enfin, la procédure **allerÀ** permet de préciser d'une façon absolue le numéro de la case à placer face au curseur. L'algorithme suivant en donne la spécification.

*Algorithme 26 : Spécification de la procédure **allerÀ***

```
Algorithme allerÀ
    # Place le curseur sous la case de numéro p.
Entrée
    v : VECTEUR[T]
    p : ENTIER
précondition
    estDéfini(v)
    v.début ≤ p ≤ v.fin
postcondition
    position(v) = p
    v.t     = ancien(v).t
    v.début = ancien(v).début
    v.fin   = ancien(v).fin
fin allerÀ
```

Accès aux composantes du vecteur

On accède aux composantes en lecture avec **lire** ou en écriture avec **écrire**. La fonction **lire** rend une copie de la composante repérée par le curseur. Sa spécification est donnée par l'algorithme suivant.

*Algorithme 27 : Spécification de **lire***

```
Algorithme lire
    # La composante repérée par le curseur.
Entrée
    v : VECTEUR[T]
```


© ENI Editions - All rigths reserved

```
Résultat : T
précondition
    estDéfini(v)
    non estAvant(v)
    non estAprès(v)
postcondition
    Résultat = v.t[position(v)]
fin lire
```

La procédure **écrire** place, dans la case repérée par le curseur, l'élément précisé en paramètre. Le contenu de la case avant cette opération est perdu. La spécification de cette procédure est la suivante :

*Algorithme 28 : Spécification de la procédure **écrire***

```
Algorithme écrire
    # Placer e dans la case repérée par le curseur.
Entrée
    v : VECTEUR[T]
    e : T
précondition
    estDéfini(v)
    non estAvant(v)
    non estAprès(v)
postcondition
    lire(v) = e
    (Les autres cases ne sont pas modifiées)
fin écrire
```

On peut alors passer à la solution de la deuxième question.

On veut déterminer la valeur de la composante minimum d'un vecteur. Soit **min** cette fonction. Sa spécification est la suivante :

*Algorithme 29 : Spécification de la fonction **min***

```
Algorithme min
    # La composante minimum de vecteur.t[début .. fin].
Entrée
    vecteur : VECTEUR[T → COMPARABLE]
Résultat : T
précondition
    estDéfini(vecteur)
postcondition
    vecteur.début = vecteur.fin => Résultat =
        lire(allerÀinv(vecteur, vecteur.début))
    vecteur.début < vecteur.fin => Résultat =
        inf
            (
            lire(allerÀinv(vecteur, vecteur.début)),
            min(copie(vecteur, vecteur.début+1, vecteur.fin))
            )
fin min
```

La fonction **allerÀinv** rend une copie du vecteur, mais avec un curseur positionné sous la case dont on donne le numéro en deuxième paramètre. La fonction **inf** a déjà été définie au chapitre Itération. La fonction **copie** réalise une copie du vecteur entre les numéros des cases indiqués. La spécification de **min** précise que le résultat est la composante lue dans la case de numéro `début` si elle est la seule du vecteur. Si elle n'est pas la seule, le résultat est la plus petite des composantes entre la première et le **min** du reste du vecteur. C'est donc la même définition, aux notations près, que celle de l'algorithme de même nature étudié au chapitre Itération.

La réalisation utilise les opérations définies sur un vecteur. Comme `début ≤ fin`, le vecteur possède au moins une composante. Il suffit donc de le parcourir à partir de cette composante. Le variant de contrôle se déduit simplement de la version étudiée au chapitre Itération. Pour l'invariant, il en est de même, mais on ne peut pas utiliser les procédures déjà définies à cause des effets de bord : elles modifient le vecteur. D'où l'utilisation de **précédentInv**, dans l'algorithme ci-dessous, qui rend une copie du vecteur dans laquelle le curseur est placé sous la case qui précède celle sous laquelle se trouve le curseur du vecteur d'origine. On obtient alors la solution suivante, dans laquelle on ne répète pas la pré et la postcondition.

© ENI Editions - All rigths reserved

*Algorithme 30 : Réalisation de la fonction **min***

```
Algorithme min
    # La composante minimum de vecteur.t[début .. fin].
...
variable
    e : T
    v : VECTEUR[T → COMPARABLE][vecteur.début, vecteur.fin]
initialisation
    v <- copie(vecteur, vecteur.début, vecteur.fin)
    allerÀ(v, début)
    Résultat <- lire(v)
    suivant(v)
jusqu'à
    position(v) > v.fin
    invariant
        position(v) > v.début => Résultat =
            min(
                copie
                    (
                    vecteur,
                    vecteur.début+1,
                    précédentInv(vecteur)
                    )
                )
    variant de contrôle
        v.fin - position(v) + 1
répéter
    e <- lire(v)
    si
        Résultat > e
    alors
        Résultat <- e
    fin si
    suivant(v)
fin répéter
...
fin min
```

L'exercice suivant complète les exercices résolus précédemment.

Exercice 4 : Opérations sur les vecteurs et les vecteurs invariables

*1. Écrire la fonction **copie** qui réalise la copie d'un vecteur entre les composantes de numéros précisés en paramètres.*

*2. Écrire les opérations **précédentInv** et **allerÀinv** utilisées dans les spécifications précédentes.*

3. Compléter par un jeu exhaustif des versions des fonctions, mais sur les vecteurs invariables.

*4. Écrire les algorithmes **rangDuMin**, **max** et **rangDuMax** pour les vecteurs.*

*La procédure **écrire** dans un vecteur n'a pas été complètement spécifiée. Il reste à donner un sens algorithmique ou mathématique à l'expression « les autres composantes ne sont pas modifiées ».*

*5. Écrire les spécifications complètes de la procédure **écrire** dans un vecteur.*

2. Fusion de deux vecteurs triés

La fusion des vecteurs procède à la lecture des composantes. Celles d'un vecteur donné sont lues et enregistrées dans le vecteur résultat tant qu'elles restent inférieures à la composante courante de l'autre vecteur. La difficulté vient de ce que les deux vecteurs n'ont pas nécessairement la même taille. Un vecteur étant épuisé, il faut recopier l'autre. Soient vt1 et vt2, les deux vecteurs à fusionner, et vr le vecteur résultat. Une version difficile à écrire utilise deux itérations dans les alternants d'une alternative sur la comparaison des composantes courantes de vt1 et vt2. Cette version est laissée en exercice. La version qui est proposée ici est plus lisible et plus simple à comprendre.

La lecture ne peut se faire que dans un vecteur où le curseur est placé entre la première et la dernière composante.

© ENI Editions - All rigths reserved

Lorsque le curseur est avant la première ou après la dernière, la lecture n'est pas définie. Soit **lireMax**, la fonction de lecture d'un vecteur qui rend INFINI lors d'une tentative de lecture en dehors des limites du vecteur. La spécification de cette fonction est donnée par l'algorithme suivant.

*Algorithme 31 : Spécification de **lireMax***

```
Algorithme lireMax
# La composante sous le curseur ou INFINI pour une lecture hors
# limites.
Entrée
    v : VECTEUR[T]
Résultat : T
précondition
    estDéfini(v)
    non estAvant(v)
postcondition
    estAprès(v)     => Résultat = INFINI
    non estAprès(v) => Résultat = lire(v)
fin lireMax
```

Ainsi, on peut toujours **avancer** et **lire** dans un vecteur. Pour fusionner les deux vecteurs v1 et v2, on les parcourt en appelant **lireMax** sur celui dont les valeurs des composantes lues restent inférieures aux valeurs des composantes de l'autre vecteur.

La spécification de l'algorithme de fusion est étudiée dans la section suivante.

a. Spécification de l'algorithme de fusion

L'algorithme reçoit, en entrée, les deux vecteurs à fusionner et les numéros des composantes extrêmes de chacun d'eux sur lesquelles porte la fusion. Il calcule le vecteur résultat vr qu'il retourne en paramètre de sortie. On suppose que le vecteur résultat est défini par l'algorithme qui appelle les services de l'algorithme de fusion. Nous pouvons donc préciser la signature de cette procédure.

*Algorithme 32 : Signature de **fusionner***

```
Algorithme fusionner
    # Fusionner v1 et v2 dans vr en ordre croissant.
Entrée
    v1, v2, vr : VECTEUR[T → COMPARABLE]
...
```

Les vecteurs à fusionner sont aussi des paramètres en sortie car on ne garantit pas que les curseurs ne seront pas modifiés.

La précondition énonce d'abord les contraintes habituelles sur les vecteurs. De plus, les deux vecteurs en entrée sont triés en ordre croissant et le nombre de composantes déclarées pour le résultat est au moins la somme des nombres de composantes de chacun des vecteurs à fusionner. La précondition s'écrit alors :

```
...
précondition
    estDéfini(v1)  ; estDéfini(v2) ; estDefini(vr)
    estTriéAsc(v1) ; estTriéAsc(v2)
    cardinal(vr) = cardinal(v1) + cardinal(v2)
...
```

La postcondition exprime que `vr` est trié en ordre croissant. Les tableaux `v1.t` et `v2.t` ne sont pas modifiés. Par contre, on doit dire explicitement que tous les éléments des vecteurs en entrée se retrouvent, sans exception, dans le vecteur résultat. Autrement dit, indépendamment de l'ordre des éléments dans `vr`, le tableau `vr.t` est la *réunion*, au sens de la réunion des multi-ensembles, de `v1.t` et `v2.t`. La postcondition est alors la suivante :

```
...
postcondition
    estTriéAsc(vr)
    estÉgal(vr, union(v1, v2))
    v1.t = ancien(v1).t
    v1.début = ancien(v1.début) ; v1.fin = ancien(v1.fin)
```

© ENI Editions - All rigths reserved

```
    v2.t = ancien(v2).t
    v2.début = ancien(v2.début) ; v2.fin = ancien(v2.fin)
...
```

Nous pouvons passer à l'étude de la fusion.

b. Analyse de la fusion

On lit deux valeurs, V1 depuis v1 et V2 depuis v2 à la position du curseur jusqu'à ce que les deux vecteurs soient épuisés. On peut donc poser en hypothèse :

Faire une hypothèse sur l'état actuel

Hypothèse (H) : on a lu V1 de v1 et V2 de v2.

Voir si c'est fini

C'est fini lorsque les lectures ont obtenu INFINI puisqu'alors, les vecteurs sont épuisés :

```
...
C'est fini lorsque V1 = V2 = INFINI
...
```

Sinon, c'est que l'une au moins de ces valeurs est une composante valide à enregistrer dans le vecteur résultat `vr`.

Se rapprocher de l'état final

```
...
si
    V1 ≤ V2
alors
    enregistrer V1 dans vr
    relire v1
sinon
    enregistrer V2 dans vr
    relire v2
fin si
...
```

On peut alors passer à la position suivante de `vr` et recommencer.

Initialiser le calcul

Il s'agit de placer le système dans un état où une valeur a déjà été lue depuis chaque vecteur. Il faut donc positionner les curseurs et réaliser une lecture :

```
...
initialisation
    # Placer les curseurs sous les premières composantes.
    premier(v1) ; premier(v2) ; premier(vr)
    # Effectuer une première lecture des vecteurs à fusionner.
    V1 <- lireMax(v1) ; V2 <- lireMax(v2)
...
```

On peut alors réaliser les traitements jusqu'à ce que les deux vecteurs soient épuisés.

Rédiger l'algorithme définitif

On obtient l'algorithme ci-dessous dans lequel on ne répète ni la précondition et la postcondition, ni les initialisations.

Algorithme 33 : Fusion de deux vecteurs triés - Réalisation

© ENI Editions - All rigths reserved

```
...
variable
    V1, V2 : T
initialisation
...
jusqu'à
    V1 = INFINI et V2 = INFINI
répéter
    si
        V1 ≤ V2
    alors
        écrire(vr, V1)
        suivant(v1)
        V1 <- lireMax(v1)
    sinon
        écrire(vr, V2)
        suivant(v2)
        V2 <- lireMax(v2)
    fin si
    suivant(vr)
fin répéter
...
```

L'exercice suivant propose une autre version de cet algorithme.

Exercice 5 : Fusion de deux vecteurs triés - Version 2.0

La version précédente n'est pas entièrement spécifiée. En particulier, il y manque l'invariant et le variant de contrôle.

1. Spécifier complètement l'algorithme.

*2. Étudier une version qui n'utilise pas la fonction **lireMax**.*

© ENI Editions - All rigths reserved

Exercices

1. Tri par insertion dichotomique

Les tris présentés dans la section Quelques algorithmes simples n'ont pas utilisé le secours d'un tableau supplémentaire pour construire le résultat. Ils modifient le tableau à trier pour réordonner ses composantes. Nous étudions ici une solution différente, qui construit un nouveau tableau sans modifier le tableau à trier.

Exercice 6 : Tri par insertion dichotomique

*Soit t un tableau de **T** à trier. On définit d'abord un nouveau tableau r de même cardinal. Chaque composante de t est insérée à sa place dans r, en recherchant la position d'insertion par l'algorithme **dichotomie** étudié au chapitre Itération.*

1. Écrire d'abord les spécifications de l'algorithme.

Soigner tout particulièrement la pré et la postcondition. Elles ne sont pas faciles à obtenir, mais elles fournissent un guide utile pour la construction de l'algorithme.

2. Écrire l'algorithme de tri d'un tableau par insertion dichotomique.

2. Un tri topologique

On considère n tâches t1, t2… tn soumises à des contraintes de précédence. Autrement dit, certaines tâches doivent être terminées avant de pouvoir commencer d'autres tâches. Ainsi, par exemple, on doit d'abord préparer les fondations d'une maison avant de monter les murs et les cloisons. L'exercice suivant propose de calculer un ordonnancement de tâches soumises à des contraintes de précédence.

Exercice 7 : Tri topologique

Une contrainte est exprimée à l'aide d'un couple (i,j) d'entiers compris entre 1 et n, qui indique que la tâche ti précède la tâche tj. Autrement dit, la tâche ti doit être terminée avant de commencer la tâche tj. La relation binaire « ... précède ... » ainsi définie sur l'ensemble des n tâches est une relation d'ordre partiel : certaines tâches ne sont pas comparables.

1. Faire un algorithme qui calcule un ordonnancement des n tâches satisfaisant aux contraintes.

Il est clair que les contraintes peuvent ne pas être toutes satisfaites. Dans ce cas, il n'existe pas d'ordonnancement des tâches. L'algorithme devra traiter correctement ce cas.

3. Compléter les spécifications

La section Quelques algorithmes simples a utilisé un prédicat **estExploré** qui n'a pas été complètement spécifié. En particulier, nous avons écrit : « *chaque composante de t[début .. fin] est placée après la plus grande suite de composantes dont elle est le max* ». L'exercice suivant propose de compléter cette spécification. C'est un problème difficile.

*Exercice 8 : Spécification de **estExploré***

*Écrire les spécifications du prédicat **estExploré**.*

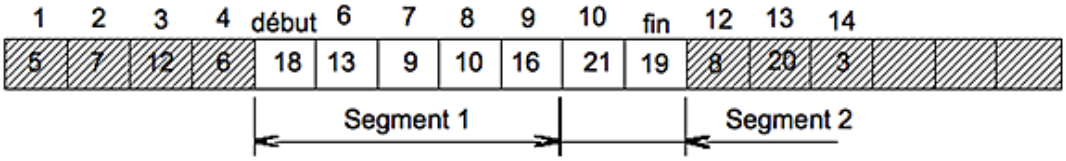
Les spécifications sont exprimées en utilisant la définition d'un segment de tableau. C'est une suite de composantes qui commence par le maximum de la suite.

Définition

*On appelle segment dans un tableau t de composantes de type **T** qui dérive de **COMPARABLE**, une plus grande suite de composantes consécutives dont la valeur maximum est la première de la suite.*

La figure ci-dessous représente un tableau et ses segments.

© ENI Editions - All rigths reserved



La partie `t[début..fin]` contient deux segments. Le premier en `t[5 .. 9]` a pour composante maximum 18 dans la case `t[5]`. Le second en `t[10 .. 11]` a pour composante maximum 21 en `t[10]`.

Le tableau `t[début .. fin]` contient `k` segments `S1, S2, …, Sk`. Chaque segment `Si` a une première composante de numéro `di` et une dernière composante de numéro `fi` qui vérifient :

$$(1 \leq i \leq k)\ (Si = t[di\ ..\ fi])$$
$$\max_{1 \leq i \leq k}(Si) = t[di]$$
$$\max_{1 \leq i \leq k}(Si) \leq t[fi + 1]$$

Explorer consiste, pour chaque segment Si, à :

- sauvegarder le max du segment : mi <- t[di] ;
- décaler les éléments de t[di+1 .. fi] d'une case vers la gauche ;
- placer le plus grand élément du segment « en haut » : t[fi] <- mi.

© ENI Editions - All rigths reserved

Notes bibliographiques

L'étude des algorithmes de tri est une étape obligée dans l'apprentissage de la programmation. Tous les livres qui traitent de l'algorithmique et des structures de données abordent le sujet, d'ailleurs avec plus ou moins de bonheur. Le plus souvent, les algorithmes présentés dans les livres sérieux sont accompagnés d'une étude de leur complexité algorithmique, ce qui est une évaluation de leurs performances. La référence sur ce sujet reste [KNUTH73b].

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a présenté la construction de quelques algorithmes dans le domaine du tri de données organisées en tableaux. Nous avons d'abord étudié la spécification générale d'un algorithme de tri qui n'utilise pas de tableau supplémentaire pour trier les données. Nous avons ensuite présenté la définition d'une structure sur laquelle est définie un curseur, ce qui permet d'itérer simplement sur les données. Cette structure a été utilisée pour écrire un algorithme de fusion en ordre croissant de deux tableaux triés. Des exercices ont permis d'aller plus loin dans la construction d'algorithmes, en abordant l'ordonnancement de tâches soumises à des contraintes de précédence. L'ambition de ce chapitre n'était pas l'étude du tri des données comme problème général. Toute étude sérieuse, même restreinte à quelques solutions types doit être accompagnée d'une évaluation de la complexité des algorithmes proposés, ce qui n'est pas le cas ici.

© ENI Editions - All rigths reserved

Bibliographie

[KNUTH73b] Donald KNUTH : *The Art of Computer Programming - Vol 3 : Sorting and Searching* ; ADDISON WESLEY, 1973.

© ENI Editions - All rigths reserved

Introduction

Le problème de l'édition d'un entier a déjà été abordé au chapitre Récursivité qui a traité de la récursivité. Ce court chapitre présente des exercices qui utilisent les techniques d'édition d'un entier, pour résoudre quelques problèmes simples. La section Édition d'un entier dans une base quelconque commence par résoudre complètement un exercice posé au chapitre Itération. La section Conversion d'un entier en chiffres romains étudie la conversion d'un nombre entier en chiffres romains. La section Vérification des identifiants d'entreprises présente les algorithmes qui permettent de vérifier la validité d'un identifiant d'entreprise et la section Vérification des identifiants de livres d'un identifiant de livre.

© ENI Editions - All rigths reserved

Édition d'un entier dans une base quelconque

Ce problème a déjà été posé au chapitre Récursivité dans lequel la section Vérification des identifiants d'entreprises présentait le problème et un exercice en demandait une première solution. Dans cette section, l'exercice est résolu complètement. La section Nombre de chiffres d'un entier calcule le nombre de chiffres d'un entier exprimé dans une base B quelconque. La section Résolution du problème d'édition résout le problème d'édition proprement dit.

1. Nombre de chiffres d'un entier

On donne un entier n. *Calculer le nombre de chiffres de la représentation de n en base B = dix.* Remarquer d'abord que nous pouvons nous restreindre aux entiers positifs. Le nombre de chiffres d'un entier négatif est le nombre de chiffres de sa valeur absolue.

Obtenir les chiffres d'un entier est un problème qui sera résolu à la section suivante. Cependant, nous pouvons déjà voir qu'ils peuvent être successivement obtenus par une suite de divisions par B = dix. Le nombre n est divisé puis, successivement, tous les quotients obtenus, jusqu'à un quotient nul. Le nombre de chiffres est le nombre de divisions. Voyons cela sur un exemple.

Exemple

Calculer le nombre de chiffres de 325 exprimé en base dix.

Il suffit de diviser 325 par 10 :

- la première division donne 32 et il reste 5 ;
- la seconde division donne 3 et il reste 2 ;
- la troisième division donne 0 et il reste 3.

Trois divisions ont permis d'atteindre un quotient nul. Le nombre de chiffres cherché est 3.

Remarquer aussi qu'il est possible d'arrêter l'itération dès que le quotient d'une division devient strictement inférieur au diviseur.

Écrire cet algorithme est facile et nous pouvons passer directement à la solution définitive.

Algorithme 1 : Nombre de chiffres d'un entier en base dix - Version 1

```
Algorithme nbChiffres1
    # Nombre de chiffres de la représentation de n en base dix.
Entrée
    n : ENTIER
Résultat : ENTIER
variable
    dividende : ENTIER <- abs(n)
    diviseur  : ENTIER <- 10
initialisation
    Résultat <- 1 # Au moins un chiffre par entier.
Jusqu'à
    dividende < diviseur
    invariant
        n = 0 => Résultat = 1
        n ≠ 0 et dividende > 0 =>
                Résultat =  log(abs(n)  -  log(dividende)  + 1
    variant de contrôle
        dividende
répéter
    dividende <- quotient(dividende, diviseur)
    Résultat  <- Résultat + 1
fin répéter
postcondition
    n = 0 => Résultat = 1
```

© ENI Editions - All rigths reserved

```
    n ≠ 0 => Résultat - 1 ≤ log(abs(n)) < Résultat
fin nbChiffres1
```

La notation ⌊x⌋ est mise pour « le plus grand entier inférieur ou égal à ». Lorsque x est positif, c'est sa partie entière. La fonction **logarithme** utilisée rend le logarithme en base dix de son argument. La distinction des deux cas, suivant que n est nul ou non, est nécessaire puisque le logarithme n'est défini que pour les valeurs positives de son paramètre.

Exercice 1 : Nombre de chiffres d'un entier - généralisation

On peut faire l'économie des fonctions mathématiques utilisées pour spécifier en développant d'abord une version récursive de cet algorithme.

*1. Écrire une version récursive de l'algorithme **nbChiffres1** précédent. Utiliser alors cette nouvelle version pour spécifier la première version.*

2. Écrire de même l'algorithme qui calcule le nombre de chiffres d'un entier exprimé dans une base B quelconque.

Le nombre B ne peut pas être quelconque et il faudra donc préciser le problème.

2. Résolution du problème d'édition

La section précédente a montré la voie pour éditer un entier. Il s'agit d'obtenir ses chiffres et de les convertir en caractères. Obtenir les chiffres d'un entier exprimé dans une base donnée consiste à effectuer une suite de divisions par la base. Les chiffres cherchés sont les restes des divisions, mais convertis en caractères. L'itération termine lorsque le quotient devient strictement inférieur au diviseur. La seule difficulté de principe qui subsiste est la conversion en caractères. Comme on ne dispose que de 26 lettres et 10 chiffres, la base de numération doit rester inférieure à 36.

Cette affirmation n'est pas tout à fait exacte. On peut représenter un chiffre d'une base B > 36 par l'expression de sa valeur en base dix. Pensez, par exemple, à la représentation d'un entier sur 32 bits en base deux cent cinquante-six. C'est cette représentation qui est utilisée, par exemple, pour noter les adresses réseau en IPv4. Ainsi, une adresse comme 172.10.23.5 par exemple représente, en base deux cent cinquante-six, le nombre $172x256^3 + 10x256^2 + 23x256 + 5 = 2886342405$ en base dix.

Pour assurer la conversion, on utilise un tableau qui donne les chiffres obtenus par division et, pour chacun d'eux, le caractère qui le représente. Voici, par exemple, le tableau de conversion en base seize.

Entier	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chiffre	' 0'	' 1'	' 2'	' 3'	' 4'	' 5'	' 6'	' 7'	' 8'	' 9'	'A'	'B'	'C'	'D'	'E'	'F'

La première ligne donne les nombres obtenus comme restes des divisions et la seconde donne les caractères qui les représentent en base seize. *Comment assurer cette conversion* ?

Nous avons vu, au chapitre Structures élémentaires, la fonction **code** qui prend en paramètre un caractère et qui rend le code numérique qui le représente dans le jeu de caractères utilisés. La fonction **caractère** est la fonction réciproque. Elle prend en entrée un code numérique et elle rend le caractère associé à ce code dans le jeu de caractères utilisés. Pour obtenir le caractère du tableau ci-dessus qui représente un entier n > 9 en base seize, il suffit donc de calculer :

```
caractère(code('A') + n - 10)
```

Lorsque l'entier est inférieur à 10, 'A' est remplacé par '0' (le caractère zéro) et `n - 10` par n dans cette formule. Soit **chiffre** la fonction à définir qui réalise ces opérations. Écrire une première version de l'algorithme qui calcule la représentation d'un entier en base dix est donc simple. C'est fait par l'algorithme suivant :

Algorithme 2 : Conversion d'un entier en base dix

```
Algorithme conversion2
    # La représentation en chiffres d'un entier en base dix.
Entrée
    n : ENTIER# L'entier à convertir.
Résultat : CHAÎNE
précondition
    n ≥ 0
constante
```


© ENI Editions - All rigths reserved

```
    SÉPARATEUR : CHAÎNE <- ':' # Caractère séparateur des chiffres.
    BASE       : ENTIER <- 10  # La base de conversion.
variable
    dividende  : ENTIER <- n
    r, q       : ENTIER # Reste et quotient dans une division.
initialisation
    Résultat <- CHAÎNE_VIDE
jusqu'à
    dividende < BASE
    invariant
        ...
    variant de contrôle
        ...
répéter
    q <- quotient(dividende, BASE)
    r <- reste(dividende, BASE)
    Résultat <- SÉPARATEUR ⊕ chaîne(chiffre(r)) ⊕ Résultat
    dividende <- q
fin répéter
Résultat <- chaîne(chiffre(dividende)) ⊕ Résultat
postcondition
    ...
fin conversion2
```

Le séparateur est utilisé pour séparer les chiffres du résultat et mettre ainsi en évidence la conversion réalisée. On obtient de cette façon '3:2:5' pour la conversion de 325 et '0' pour la conversion de 0. L'algorithme proposé est volontairement incomplet. En particulier, le résultat est donné sans en construire une analyse raisonnée et certains détails de réalisation ne sont pas triviaux. L'exercice suivant propose de compléter cet algorithme.

Exercice 2 : Conversion d'un entier en base dix

1. Écrire une version récursive qui permettra de terminer la spécification de l'algorithme.

2. Développer l'analyse complète de l'algorithme. En particulier, compléter avec l'invariant et le variant de contrôle.

Il faut vérifier soigneusement chacune des étapes de la solution. Il n'est pas rare de voir des solutions fausses pour cet exercice, notamment en ce qui concerne les cas limites.

3. Modifier cet algorithme pour relâcher la contrainte imposée par la précondition.

4. Écrire un algorithme qui prend en entrée le nombre n à convertir et la base B de conversion. Préciser le problème et étudier les structures de données nécessaires.

5. Étendre les solutions au cas des nombres entiers négatifs.

3. Résolution du problème réciproque

Le problème réciproque s'énonce simplement. On donne une chaîne de caractères qui représente un nombre entier, disons positif dans un premier temps, dans une base B. *Étudier un algorithme qui calcule la valeur de ce nombre.*

Exemple

Convertir la chaîne de caractères '2753' exprimée en base huit.

On sait convertir chacun des caractères pour obtenir l'entier qu'il représente en base dix. Ainsi, on écrira :

```
c <- nombre('7')
```

pour convertir le caractère '7' et obtenir le nombre `c = 7`, ici en base dix. Nous savons donc calculer la suite des nombres associés à chacun des caractères de la chaîne. Pour obtenir le nombre qu'elle représente, il suffit alors de calculer :

$2753_8 = (2x8^3) + (7x8^2) + (5x8^1) + (3x8^0) = 1515_{10}$

Plus généralement, soit s une chaîne de caractères qui représente un nombre entier positif n exprimé en base B > 1. Avec $s = a_k a_{k-1} \ldots a_1 a_0$ où les a_i sont les caractères, le nombre n est obtenu par :

© ENI Editions - All rigths reserved

$n = \mathbf{nombre}(a_k) \times B^k + \mathbf{nombre}(a_{k-1}) \times B^{k-1} + \ldots + \mathbf{nombre}(a_0) \times B^0$

Au lieu de calculer des exponentiations (élévation à une puissance), on peut utiliser une factorisation qui permet de ne faire que des multiplications et des additions. Ainsi, pour le nombre de l'exemple précédent, on calculera :

$2753_8 = 8x(8x(8x2 + 7) + 5) + 3 = 1515_{10}$

Il est plus simple et efficace de commencer par la fin de la chaîne :

$2753_8 = 3 + 8x(5 + 8x(7 + 8x2))) = 1515_{10}$

Cette factorisation est appelée la « factorisation de HÖRNER ».

Soit donc **dernier** la fonction qui rend le dernier caractère de la chaîne qu'elle reçoit en paramètre. Sa spécification est donnée ci-dessous.

*Algorithme 3 : Spécification de la fonction **dernier***

```
Algorithme dernier
    # Le dernier caractère de ch ou CAR_VIDE.
Entrée
    ch : CHAÎNE
Résultat : CARACTÈRE
précondition
    estDéfinie(ch)
postcondition
    longueur(ch) = 0 => Résultat = CAR_VIDE
    longueur(ch) > 0 => Résultat = item(ch, index_max(ch))
fin dernier
```

La fonction **item** a été définie au chapitre Structures élémentaires. Elle retourne le caractère de son premier paramètre qui occupe le rang ayant pour valeur celle de son second paramètre.

Lorsque le dernier caractère a été converti, il reste à convertir le début de la chaîne, c'est-à-dire toute la chaîne, privée de son dernier caractère. Soit **début** la fonction qui rend une copie de la chaîne qu'elle reçoit en paramètre, mais privée de son dernier caractère. Sa spécification est donnée par l'algorithme ci-dessous.

*Algorithme 4 : Spécification de la fonction **début***

```
Algorithme début
    # Copie de ch privée de son dernier caractère.
Entrée
    ch : CHAÎNE
Résultat : CHAÎNE
précondition
    estDefinie(ch)
postcondition
    longueur(ch) = 0 => Résultat = CHAÎNE_VIDE
    longueur(ch) > 0 => Résultat = inverse(fin(inverse(ch)))
fin début
```

Cette spécification exprime que le début de la chaîne est l'inverse de la fin de son inverse. La fonction **fin** a été définie au chapitre Récursivité. Elle retourne une copie de la chaîne qu'elle reçoit en paramètre, mais privée de son premier caractère.

Nous pouvons alors écrire la spécification de la fonction de conversion d'une chaîne en un entier. C'est l'algorithme suivant :

Algorithme 5 : Spécification de la fonction de conversion d'une chaîne en entier

```
Algorithme conversion3
    # L'entier dont ch représente la valeur en base B.
Entrée
    ch : CHAÎNE # La chaîne à convertir.
    B  : ENTIER # La base de numération.
Résultat : ENTIER
```


© ENI Editions - All rigths reserved

```
précondition
    estDéfinie(ch)
    longueur(ch) > 0
    1 < B < 37
    Les caractères de ch expriment un nombre en base B.
postcondition
    longueur(ch) = 1 => Résultat = nombre(premier(ch))
    longueur(ch) > 1 => Résultat =
            nombre(premier(ch)) + B x conversion3(début(ch))
fin conversion3
```

L'appel à la fonction **premier** dans la première clause de la postcondition est nécessaire puisque `ch` est une **CHAÎNE** alors que `nombre` prend un **CARACTÈRE** en paramètre.

*Exercice 3 : Étude de la fonction **conversion3***

*1. Écrire les réalisations itératives des fonctions **dernier** et **début**.*

La précondition exprime en Français que les caractères de `ch` doivent être des caractères « valides » pour exprimer un entier dans la base B.

2. Écrire la spécification et la réalisation d'un prédicat permettant d'écrire cette clause.

*3. Étudier une version itérative de la fonction **conversion3**.*

4. Donner les spécifications et les réalisations complètes des prédicats nécessaires pour exprimer les invariants des itérations.

© ENI Editions - All rigths reserved

Conversion d'un entier en chiffres romains

1. Conversion chiffres romains → nombre entier

Considérons d'abord le problème réciproque : on donne un nombre entier exprimé en chiffres romains. *Écrire un algorithme qui donne la valeur numérique de ce nombre.* Comme souvent, le problème doit être précisé.

La donnée est un nombre entier exprimé en chiffres romains. C'est donc une chaîne de caractères. Nous voulons obtenir sa valeur en base dix. Celle-ci peut être exprimée par une chaîne de caractères, la suite des chiffres qui représentent cette valeur en base dix, ou par un nombre entier. Convenons d'abord que nous souhaitons obtenir le nombre entier et non pas sa représentation en chiffres. Un exercice à venir propose d'écrire d'autres versions de cette conversion et, notamment, la conversion en chiffres de la base dix.

Soit *romain* la chaîne de caractères qui représente en chiffres romains le nombre à convertir. La spécification de la fonction de conversion est la suivante :

Algorithme 6 : Conversion chiffres romains → nombre entier - Version 1.0

```
Algorithme convertir1
    # L'entier en base dix égal à romain.
Entrée
    romain : CHAÎNE
Résultat : ENTIER
précondition
    estDéfinie(romain)
    romain est une représentation valide d'un entier en chiffres
    romains
postcondition
    Résultat = (l'entier égal à romain)
fin convertir1
```

Nous connaissons les valeurs associées à un chiffre romain isolé. Ainsi, 'C' correspond à 100, 'X' correspond à 10, etc. Nous savons aussi que, quand un caractère en précède un autre de valeur supérieure, il est compté négativement. Ainsi, 'IX' vaut 9 alors que 'XI' représente 11. De même, 'XM' représente 900 puisque 'M' représente 1000. Par conséquent, nous savons calculer la valeur décimale d'un entier représenté par un chiffre romain r isolé. Cette valeur décimale est obtenue par la fonction **valeur1** dont la spécification est :

*Algorithme 7 : Spécification de la fonction **valeur1***

```
Algorithme valeur1
    # Valeur entière du chiffre romain r.
Entrée
    r : CARACTÈRE
Résultat : ENTIER
précondition
    # r est un chiffre romain valide.
    r ∈ {'M', 'D', 'C', 'L', 'X', 'V', 'I'}
postcondition
    r = 'M' => Résultat = 1000
    r = 'D' => Résultat =  500
    r = 'C' => Résultat =  100
    r = 'L' => Résultat =   50
    r = 'X' => Résultat =   10
    r = 'V' => Résultat =    5
    r = 'I' => Résultat =    1
fin valeur1
```

Lorsque la valeur d'un caractère romain est obtenue, il suffit d'ajouter cette valeur au résultat déjà construit puis de passer au caractère suivant. Considérons, par exemple, la chaîne de caractères :

```
romain = 'MCDXII'
```

et l'état dans lequel aucun caractère n'a été converti. On a d'abord :

© ENI Editions - All rigths reserved

```
Résultat <- valeur1('M')
```

Mais alors, nous ne pouvons pas écrire :

```
Résultat <- Résultat + valeur1('C') # Instruction incorrecte.
```

En effet, 'C' précède 'D' et donc 'CD' représente 400 puisqu'un caractère qui précède un caractère représentant une valeur supérieure doit être retranché et non ajouté au résultat. Ainsi, ce qui est cumulé dans **Résultat** n'est pas la valeur entière représentée par le caractère. La valeur rendue par la fonction **valeur1** est la *valeur absolue* du nombre que représente le caractère. La règle de position des chiffres romains permet ensuite de déterminer la valeur effective signée représentée par le caractère. Le signe du nombre déduit de cette valeur est obtenu en observant le caractère qui suit celui qui est analysé :

```
...
v <- valeur1(caractère analysé)
si
    v < valeur1(caractère suivant)
alors
    # Application de la règle de position.
    v <- -v
fin si
Résultat <- Résultat + v
...
```

À ce stade, un caractère vient d'être converti. Il ne reste plus qu'à convertir le reste de la chaîne :

```
...
Résultat <- Résultat + convertir1(fin(romain))
...
```

Essayons cet algorithme sur la chaîne romain = 'MCDXII' :

```
(1a) v <- valeur1('M') => v <- 1000
(1b) v = 1000 > valeur1('C') => v = 1000
(1c) Résultat <- Résultat + v => Résultat = 1000
(1d) Résultat <- 1000 + convertir('CDXII')
```

L'appel récursif à **convertir1** a le même effet :

```
(2a) v <- valeur1('C') => v <- 100
(2b) v = 100  < valeur1('D') => v = -100
(2c) Résultat <- Résultat + v => Résultat = 900
(2d) Résultat <- 900 + convertir('DXII')
```

Le nouvel appel convertit alors 'D' :

```
(3a) v <- valeur1('D') => v <- 500
(3b) v = 500 > valeur1('X') => v = 500
(3c) Résultat <- Résultat + v => Résultat = 1400
(3d) Résultat <- 1400 + convertir('XII')
```

Le reste de la chaîne est converti de la même façon et nous obtenons 1412. Nous pouvons donc écrire une première version de cet algorithme :

Algorithme 8 : Conversion chiffres romain → nombre entier - Version 1.0

```
Algorithme convertir1
    # L'entier en base dix égal à romain.
Entrée
    romain : CHAÎNE
Résultat : ENTIER
précondition
    estDéfinie(romain)
    longueur(romain) > 0
    romain est une représentation valide d'un entier en chiffres
```


© ENI Editions - All rigths reserved

```
    romains
réalisation
    si
        longueur(romain) = 1
    alors
        Résultat <- valeur1(premier(romain))
    sinon si
        valeur1(premier(romain)) < valeur1(item(romain,2))
    alors
        Résultat <- convertir1(fin(romain)) -
                    valeur1(premier(romain))
    sinon
        Résultat <- convertir1(fin(romain)) +
                    valeur1(premier(romain))
    fin si
postcondition
    Résultat = (l'entier égal à romain)
fin convertir1
```

L'utilisation de la fonction **premier**, qui retourne le premier caractère de son argument, est nécessaire puisque **valeur1** prend en paramètre un **CARACTÈRE** alors que `romain` est une **CHAÎNE**.

Étudions à présent une version itérative de **convertir1**. Ce sera **convertir2**.

L'algorithme précédent consiste à convertir le premier caractère en le comparant au second. Dans une version itérative, on convertit un caractère quelconque en le comparant à celui qui le suit.

Faire une hypothèse sur l'état actuel

Le dernier caractère converti est le caractère de rang `i - 1` de la chaîne `romain`. Le caractère qui le suit est **item** (romain, i).

Hypothèse (H) : romain[1 .. i - 1] a été converti dans Résultat. La valeur suivante est vs = valeur1(item(romain, i)).

Voir si c'est fini

C'est fini lorsque i = **longueur**(romain). Mais lorsque c'est fini, vs contient une valeur qui n'a pas encore été cumulée dans **Résultat**.

```
...
si
    longueur(romain) = i
alors
    Résultat <- Résultat + vs
    c'est fini
sinon
...
```

Se rapprocher de l'état final

On se rapproche de l'état final en calculant la valeur du caractère suivant. Sa valeur absolue est vs. On obtient sa valeur définitive en comparant cette valeur absolue à celle du caractère qui le suit :

```
...
sinon
    # Déterminer la valeur absolue du caractère suivant.
    v <- vs
    vs <- valeur1(item(romain, i+1))
    # Appliquer la règle de position des chiffres romains.
    si
        v < vs
    alors
        v <- -v
    fin si
    Résultat ← Résultat + v
...
```

© ENI Editions - All rigths reserved

L'état obtenu est alors décrit par l'assertion suivante :

```
...
assertion
    Résultat = (résultat de la conversion de romain[1 .. i])
    vs = valeur1(item(romain, i+1))
...
```

On retrouve l'état décrit par **(H)** en avançant `i` :

```
...
i <- i + 1
...
```

Initialiser le calcul

Dans l'état initial, aucun caractère n'a encore été converti et donc **Résultat** = 0. Le prochain caractère à observer est celui de rang `i` et, initialement, c'est le premier : `i <- 1`. Enfin, la valeur suivante est celle du premier caractère et donc :

```
vs <- valeur1(item(romain, 1))
```

D'où le bloc des initialisations :

```
...
variable
    i, v, vs : ENTIER
initialisation
    Résultat <- 0
    i  <- 1
    vs <- valeur1(item(romain, 1))
...
```

Rédiger l'algorithme définitif

Nous devons pouvoir écrire l'invariant de l'itération en exprimant d'une façon algorithmique les assertions de l'hypothèse **(H)**. Celle-ci fait intervenir la conversion partielle de la chaîne jusqu'au caractère de rang `i - 1`. Posons alors **convertir4**, la fonction qui prend en paramètres la chaîne à convertir et le rang entier donnant le numéro du caractère où s'arrête la conversion. Ainsi, par exemple :

```
n <- convertir4(romain, 4)
```

ne convertit que la chaîne faite des quatre premiers caractères de `romain`. Pour convertir toute la chaîne, on devra écrire :

```
n <- convertir4(romain, longueur(romain))
```

On peut remarquer aussi que **valeur1** calcule la valeur d'un caractère de la chaîne `romain`, mais pas nécessairement le premier. Soit alors **valeur2**, la fonction qui prend en entrée une chaîne et un rang entier et qui retourne la valeur absolue du caractère qui occupe ce rang dans la chaîne. Ainsi, pour obtenir la valeur absolue du second caractère, on écrira :

```
vs <- valeur2(romain, 2)
```

Dans la suite, les fonctions **convertir4** et **valeur2** remplacent respectivement les fonctions **convertir2** et **valeur1** dont l'écriture est laissée en exercice.

Nous pouvons alors exprimer l'invariant et le variant de contrôle de l'itération :

```
...
invariant
    i = 1 => Résultat = 0 et vs = valeur2(romain, 1)
    i > 1 => Résultat = convertir4(romain, i - 1) et
                    vs = valeur2(romain, i)
variant de contrôle
```


© ENI Editions - All rigths reserved

```
    longueur(romain) - i
...
```

La spécification de l'algorithme est la suivante :

```
Algorithme convertir4
    # Convertir romain en base dix.
Entrée
    romain : CHAÎNE
Résultat : ENTIER
précondition
    estDéfinie(romain)
    'romain' est une représentation valide d'un entier en
    chiffres romains.
postcondition
    longueur(romain) = 1 => Résultat = valeur2(romain, 2)

    longueur(romain) > 1 et valeur2(romain, 1) < valeur2(romain, 2)
    => Résultat =
     convertir4(fin(romain, longueur(romain))) - valeur2(romain, 1)

    longueur(romain) ≤ 1 et valeur2(romain, 1) < valeur2(romain, 2)
    => Résultat =
     convertir4(fin(romain, longueur(romain))) + valeur2(romain, 1)
fin convertir4
```

L'algorithme qui suit donne une version complète, mais pas encore définitive, de cet algorithme.

Algorithme 9 : Conversion chiffres romains → nombre entier - Version 2.0

```
...
variable
    i, v, vs : ENTIER
initialisation
    Résultat <- 0
    i <- 1
    vs <- valeur2(romain, 1)
jusqu'à
    i ≥ longueur(romain)
    invariant
        i = 1 => Résultat = 0 et vs = valeur2(romain, 1)
        i > 1 => Résultat = convertir4(romain, i - 1) et
                        vs = valeur2(romain, i)
    variant de contrôle
        longueur(romain) - i
répéter
    v <- vs
    vs <- valeur2(romain, i + 1)
    si
        v < vs
    alors
        v <- -v
    fin si
    Résultat <- Résultat + v
    assertion
        Résultat = convertir4(romain, i) et
              vs = valeur2(romain, i + 1)
    i <- i + 1
    assertion
        Résultat = convertir4(romain, i - 1) et
              vs = valeur2(romain, i )
fin répéter
...
fin convertir4
```

© ENI Editions - All rigths reserved

Cet algorithme peut être légèrement amélioré en remarquant que certains calculs peuvent être « économisés ». C'est le cas du calcul de la longueur de la chaîne, par exemple. Il est aussi possible d'éviter les multiples calculs de `i + 1` en déplaçant l'incrémentation. On obtient :

Algorithme 10 : Conversion chiffres romains → nombre entier - Version 2.1

```
...
variable
    i, v, vs, long : ENTIER
initialisation
    Résultat <- 0
    i    <- 1
    vs   <- valeur2(romain, 1)
    long <- longueur(romain)
jusqu'à
    i ≥ long
    invariant
        i = 1 => Résultat = 0 et vs = valeur2(romain, 1)
        i > 1 => Résultat = convertir4(romain, i - 1) et
                        vs = valeur2(romain, i)
    variant de contrôle
        long - i
répéter
    i <- i + 1
    v <- vs
    vs <- valeur2(romain, i + 1)
    si
        v < vs
    alors
        v <- -v
    fin si
    Résultat <- Résultat + v
    assertion
        Résultat = convertir4(romain, i) et
              vs = valeur2(romain, i + 1)
    assertion
        Résultat = convertir4(romain, i - 1) et
              vs = valeur2(romain, i )
fin répéter
...
fin convertir4
```

L'exercice suivant propose quelques extensions à cette activité.

Exercice 4 : D'autres versions de la conversion chiffres romains → nombre entier

*1. Donner les spécifications et les réalisations des fonctions **valeur1** et **valeur2**.*

Ces deux fonctions peuvent être réalisées en utilisant une énumération. C'est probablement ce que vous avez fait pour résoudre la question précédente. Dans ce cas, résoudre la question suivante.

*2. Écrire une version des réalisations de **valeur1** et **valeur2** qui explorent un tableau pour convertir un caractère.*

3. Écrire une version de l'algorithme de conversion qui rend la représentation du nombre par des chiffres en base dix au lieu de calculer sa valeur.

La précondition de l'algorithme de conversion impose que la chaîne qui représente le nombre en chiffres romains soit valide.

4. Comment se comporte l'algorithme proposé lorsqu'un caractère qui n'est pas reconnu comme un chiffre romain valide est présent dans la chaîne paramètre ? Modifier s'il y a lieu pour relâcher cette contrainte.

Il est donc possible d'écrire une version de ces algorithmes qui se comporte correctement et qui rend le bon résultat, même si la chaîne de caractères qui représente l'entier écrit en chiffres romains est mal formée.

5. Discuter les raisons pour lesquelles cette solution n'est pas recommandée malgré sa robustesse.

*6. Il n'est pas nécessaire que la précondition impose **longueur**(romain) = 1. Modifier cet algorithme pour qu'il accepte une chaîne de longueur nulle dont la valeur sera alors 0.*

© ENI Editions - All rigths reserved

2. Conversion nombre entier → chiffres romains

Il s'agit, à présent, de traiter le problème direct. On donne un nombre entier. *Écrire l'algorithme de la fonction qui retourne sa représentation en chiffres romains*. Ce problème n'est pas fondamentalement difficile, mais un peu plus délicat que le précédent. Précisons-le d'abord.

On donne un nombre entier n, compris entre 0 et 3000 : 0 ≤ n ≤ 3000. Il s'agit d'écrire une fonction qui rend la chaîne de caractères représentant ce nombre en chiffres romains. La borne 3000 est choisie pour simplifier le problème en restreignant le nombre de caractères représentant les chiffres romains à prendre en compte. Ce qui complique le problème, c'est la règle de position. Considérons, par exemple, le cas du nombre 1912. Sans cette règle, il suffit de « l'éditer » par une suite de divisions par 1000, 500, 100, 50 et 10. On obtient alors :

```
1912 = 1000 + 500 + 4x100 + 10 + 2 ≡ 'MDCCCCXII'
```

Mais `500 + 4x100 = 900` et `900 ≡ 'CM'` et non pas `'DCCCC'`. Par conséquent, `1912 ≡ 'MCMXII'`. De même :

```
1999 = 1000 + (500 + 400)  + (50 + 40)  + (5 + 4)
     = 1000 + (1000 - 100) + (100 - 10) + (10 - 1)
     ≡ 'MCMXCIX'
```

Pour éviter les perturbations introduites par les représentations des nombres 900, 400, etc, on convient de considérer ces nombres comme des représentations en base dix de *chiffres romains composés*. La table ci-dessous donne la conversion de tous les chiffres considérés dans cette section.

Entier	1000	900	500	400	100	90	50	40	10	9	5	4	1
Chiffre	'M'	'CM'	D	'CD'	'C'	'XC'	'L'	'XL'	'X'	'IX'	'V'	'IV'	'I'

Convenons également que le nombre 0 sera représenté par une chaîne de caractères vide. L'algorithme consiste alors à diviser successivement la suite des restes des divisions par la suite des entiers de la table ci-dessus. Voyons cela sur l'exemple de `n = 1999`.

Exemple

Convertir 1999 en chiffres romains.

La suite des divisions donne les résultats suivants :

- la division par 1000 donne 1 => 'M' reste 999 ;
- la division de 999 par 900 donne 1 => 'CM' reste 99 ;
- les divisions successives de 99 par 500, 400 et 100 donnent toutes 0 et un reste égal à 99 ;
- la division de 99 par 90 donne 1 => 'XC' reste 9 ;
- les divisions successives de 9 par 50, 40 et 10 donnent toutes 0 et un reste égal à 9 ;
- la division de 9 par 9 donne 1 => 'IX' avec un reste nul, ce qui termine le calcul.

Finalement, en rassemblant tout cela, on obtient `'MCMXCIX'` qui est le résultat attendu. Évidemment, cette méthode fait beaucoup de calculs inutiles, mais elle devrait donner des résultats corrects et nous allons nous en contenter dans une première version. Cependant, le problème n'est pas toujours aussi simple.

Exemple

Convertir 203 en chiffres romains.

Le nombre 203 s'écrit `'CCIII'` en chiffres romains. Ce qui diffère ici, c'est la répétition des chiffres. Les divisions donnent 2 pour le quotient par 100 puis 3 pour le quotient par 1. Nous devons donc préparer la chaîne `'CC'` dont le nombre de caractères est égal au quotient de la division par 100. De même, le nombre de caractères de `'III'` est égal au quotient dans la division par 1. Ainsi, le quotient de chaque division donne le nombre de caractères identiques à concaténer à droite de la chaîne **Résultat** déjà obtenue. Soit **copie**, une fonction capable de préparer une telle chaîne.

© ENI Editions - All rigths reserved

Les spécifications de cette fonction sont données par l'algorithme ci-dessous.

Algorithme 11 : Spécification de la fonction copie

```
Algorithme copie
    # La chaîne égale à n occurrences de c.
Entrée
    c : CARACTÈRE
    n : ENTIER
Résultat : CHAÎNE
précondition
    n ≥ 0
postcondition
    n = 0 => Résultat = CHAÎNE_VIDE
    n > 0 => Résultat = chaîne('c') ⊕ copie(c, n - 1)
fin copie
```

Nous pouvons alors analyser le problème posé. Soit `chiffres` la matrice qui implémente la table précédente.

Faire une hypothèse sur l'état actuel

Hypothèse (H) : on a déjà divisé par les (i - 1) premiers nombres de la matrice chiffres. Le dernier reste obtenu est r. La chaîne actuelle est Résultat.

Voir si c'est fini

C'est fini lorsque les 13 positions de la matrice `chiffres` ont été utilisées ou, mieux, lorsque le reste `r` est nul.

Se rapprocher de l'état final

On se rapproche de l'état final en réalisant une division supplémentaire :

```
...
dividende <- r              # Le dernier reste devient dividende.
diviseur  <- chiffres[1][i] # Prochain diviseur.
r <- reste(dividende, diviseur) # Reste suivant.
Résultat <- Résultat ⊕ copie
                       (
                       chiffres[2][i],
                       quotient(dividende, diviseur)
                       )
i <- i + 1
...
```

Ici, le tableau `chiffres[1]` contient les nombres entiers et le tableau `chiffres[2]` contient les chiffres romains. Cependant, ceci n'est qu'un principe de résolution. En effet, les données dans les lignes de cette matrice ne sont pas de même type. Le tableau de la première ligne contient des nombres qui seront des diviseurs. Le tableau de la deuxième ligne contient des chaînes de caractères. Nous utiliserons donc deux tableaux différents au lieu d'une matrice. Le tableau `chiffres` contiendra les chaînes de caractères pour les chiffres romains composés. Le tableau `nombres` contiendra les entiers diviseurs.

Initialiser le calcul

Nous devons réaliser l'hypothèse **(H)** pour placer le système logiciel dans l'état initial. On a divisé par les `(i - 1)` premiers nombres. Initialement, aucune division n' a encore été faite et donc, le prochain nombre à utiliser est celui de numéro `i = 1`. Le reste obtenu devient le nouveau dividende. Ce dividende est initialement le nombre à convertir et donc : `r <- n`. Enfin, **Résultat** contient le résultat obtenu par les divisions précédentes. Il est donc initialement vide. D'où le bloc des initialisations :

```
...
initialisation
    Résultat <- CHAÎNE_VIDE # Résultat actuel.
    r <- n                  # Dernier reste obtenu.
    i <- 1                  # Numéro du prochain diviseur.
```


© ENI Editions - All rigths reserved

```
...
```

Rédiger l'algorithme définitif

L'algorithme est le suivant :

Algorithme 12 : Conversion d'un entier en chiffres romains - Version 1.0

```
Algorithme convertir5
    # La représentation de n en chiffres romains.
Entrée
    n : ENTIER
Résultat : CHAÎNE
précondition
    0 ≤ n ≤ 3000
variable
    # Tableaux de conversion.
    chiffres : TABLEAU[CHAÎNE][1,13]
    nombres  : TABLEAU[ENTIER][1,13]
    i        : ENTIER # Prochaine case des tableaux chiffres et
                      # nombres à adresser.
    quotient, reste, dividende, diviseur : ENTIER
initialisation
    Résultat <- CHAÎNE_VIDE # Résultat actuel.
    reste    <- n           # Dernier reste obtenu.
    i        <- 1           # Numéro du prochain diviseur.
    Initialiser les tableaux chiffres et nombres.
jusqu'à
    reste = 0
répéter
    dividende <- reste
    diviseur  <- nombres[i]
    reste     <- reste(dividende, diviseur)
    quotient  <- quotient(dividende, diviseur)
    Résultat  <- Résultat ⊕ copie(chiffres[i], quotient)
    i <- i + 1
fin répéter
postcondition
    n = 0 => Résultat = CHAÎNE_VIDE
    n > 0 => Résultat = (la représentation de n en chiffres romains)
fin convertir5
```

L'exercice suivant propose d'améliorer cet algorithme.

*Exercice 5 : Modification de l'algorithme de **convertir5***

Cet algorithme fait beaucoup d'opérations inutiles. Même si nous ne sommes pas préoccupés par les performances dans cette initiation, il est toujours bon de s'interroger sur les améliorations possibles d'une première version d'un algorithme.

Remarquer d'abord qu'une division est inutile dès lors que le reste obtenu à la division précédente est strictement inférieur au diviseur.

1. Modifier l'algorithme pour éliminer les divisions inutiles.

*Ce faisant, on ne gagne pas seulement sur les divisions. Chaque division « économisée » permet aussi de gagner un appel à la fonction **copie**. Cet appel n'est nécessaire que lorsque le quotient d'une division est supérieur à 1.*

2. Modifier l'algorithme en conséquence.

*3. Écrire la définition complète de **copie**.*

La postcondition n'est pas exprimée complètement d'une façon algorithmique. C'est une spécification difficile.

4. Exprimer la postcondition d'une façon algorithmique.

5. Étudier l'expression de l'invariant et du variant de contrôle.

© ENI Editions - All rigths reserved

Vérification des identifiants d'entreprises

Chaque entreprise est identifiée par un numéro entier de 9 chiffres en base dix du *Système Informatique pour le Répertoire de Entreprises* ou SIREN. Chaque établissement d'une entreprise est identifié, relativement à cette entreprise, par un numéro entier à 5 chiffres en base dix appelé le *Numéro Interne de Classement* ou NIC. Le nombre de 14 chiffres obtenu en regroupant, comme s'il s'agissait d'une concaténation, le numéro SIREN d'une entreprise et le NIC de l'un de ses établissements constitue le numéro du *Système Informatique pour le Répertoire des ÉTablissements* ou SIRET.

Exemple

La société PUBLITRONIC est identifiée par les nombres suivants :

```
SIREN : 319 937 454
NIC   :             000 35
SIRET : 319 937 454 000 35
```

Les identifiants sont composés selon des règles dont l'étude dépasse le cadre de cette initiation. Cependant, une règle de cohérence peut être utilisée pour vérifier la validité d'un SIREN ou d'un SIRET. L'un de ces nombres est un numéro valide s'il satisfait à la règle de contrôle suivante :

Règle de contrôle d'un SIREN ou d'un SIRET

La somme des chiffres de rang impair augmentée de la somme des doubles des chiffres de rang pair est un multiple de 10.

Les rangs sont numérotés, dans chaque cas, de la droite vers la gauche. Le premier chiffre à droite, autrement dit le chiffre des unités, a le rang 1. Lorsque le double d'un chiffre de rang pair donne un résultat supérieur ou égal à 10, ses chiffres sont additionnés. Pour les identifiants de la société PUBLITRONIC, on obtient :

```
SIREN : 319 937 454
CONTRÔLE = 4 + 2x5 + 4 + 2x7 + 3 + 2x9 + 9 + 2x1 + 3
         = 4 +  10 + 4 +   14 + 3 + 18 + 9 +   2 + 3
         = 4 +   1 + 4 +    5 + 3 +  9 + 9 +   2 + 3
         = 40 qui est un multiple de 10 ;
SIRET : 319 937 454 000 35
CONTRÔLE = 5 + 2x3 + 0 + 2x0 + 0 + 2x4 + 5 + 2x4 + 7 + 2x3 + 9 +
           2x9 + 1 + 2x3
         = 5 +   6 + 0 +   0 + 0 +   8 + 5 +   8 + 7 +   6 + 9 +
           18 + 1 +   6
         = 5 +   6 + 0 +   0 + 0 +   8 + 5 +   8 + 7 +   6 + 9 +
           9 + 1 +   6
         = 70 qui est un multiple de 10.
```

Ces deux identifiants passent donc le contrôle.

Exercice 6 : Exemples de vérifications d'identifiants d'entreprises

Les identifiants suivants sont-ils des identifiants valides ?

- *307 961 722 et 307 961 722 000 13 ;*
- *420 783 680 ;*
- *347 364 570 ;*

Le but de cette section est de définir un algorithme capable de vérifier la validité d'un numéro SIREN ou SIRET. Soit **contrôle**, le prédicat qui retourne VRAI si et seulement si la somme de contrôle d'un nombre entier candidat à être un numéro SIREN est un multiple de 10. L'algorithme ci-dessous définit ce prédicat.

Algorithme 13 : Prédicat de contrôle d'un SIREN

```
Algorithme contrôle
    # n passe-t-il le contrôle d'un SIREN ?
Entrée
```

© ENI Editions - All rigths reserved

```
    n : ENTIER
Résultat : BOOLÉEN
précondition
    n > 0
    nbChiffres1(n) = 9
réalisation
    Résultat <- (reste(sommeCtrl(n), 10) = 0)
postcondition
    Résultat =  (reste(sommeCtrl(n), 10) = 0)
fin contrôle
```

Cet algorithme utilise la fonction **sommeCtrl** qui calcule la somme de contrôle associée à son paramètre. Ce calcul décompose le nombre en chacun de ses chiffres pour lui appliquer la règle de vérification. La seule difficulté consiste à additionner les chiffres d'un résultat quand le double d'un chiffre du nombre est supérieur ou égal à 10 ; mais le double d'un chiffre prend ses valeurs dans {10 ; 12 ; 14 ; 16 ; 18}. Dans chaque cas, on obtient la somme des deux chiffres en retranchant 9. Ainsi, on a :

10 => 1 + 0 = 10 - 9 = 1

12 => 1 + 2 = 12 - 9 = 3

14 => 1 + 4 = 14 - 9 = 5

16 => 1 + 6 = 16 - 9 = 7

18 => 1 + 8 = 18 - 9 = 9

Mais il existe d'autres façons de procéder. Bien entendu, cette fonction prend en entrée un nombre entier positif mais quelconque. Sa seule responsabilité est le calcul de la somme de contrôle. Sa spécification est la suivante :

```
Algorithme sommeCtrl
    # La somme de contrôle de n.
Entrée
    n : ENTIER
Résultat : ENTIER
précondition
    n ≥ 0
postcondition
...
```

Exercice 7 : Contrôler un SIREN ou un SIRET

*1. Écrire la spécification complète de **sommeCtrl**.*

2. Écrire la réalisation récursive et la réalisation itérative de cette fonction.

La fonction `contrôle` impose un argument de 9 chiffres. Or, il serait possible de l'utiliser également pour contrôler un SIRET.

3. Modifier la fonction contrôle pour qu'elle permette AUSSI la vérification d'un SIRET.

© ENI Editions - All rigths reserved

Vérification des identifiants de livres

Un livre est identifié par un numéro appelé un *International Standard Book Number* ou ISBN. C'est un nombre de 10 chiffres qui repère la zone géographique ou linguistique d'édition, l'éditeur dans cette zone et, bien entendu, le livre chez cet éditeur. Depuis quelques années, l'introduction des codes-barres a fait évoluer cet identifiant en un nombre de 13 chiffres, le *European Article Numbering* du système global d'identification univoque d'objets.

La structure d'un ISBN est donnée dans le tableau ci-dessous sur un exemple dont la valeur est 0-8436-1072-7.

Numéro de groupe	**Préfixe d'éditeur**	**Numéro de titre**	**Contrôle**
0	8436	1072	7

Le contrôle d'un ISBN consiste à réaliser des calculs basés sur la position, c'est-à-dire sur le rang du chiffre dans le nombre, et sa valeur. Les chiffres d'un ISBN sont numérotés de 1 à 10 en commençant par le chiffre des unités, c'est-à-dire le chiffre de contrôle. Chaque chiffre est multiplié par son rang et la somme des produits ainsi obtenus est divisée par 11. L'ISBN est valide lorsque le reste de cette division est nul.

Exemple

Vérifier la validité de 0-8436-1072-7.

```
Chiffres :  0    8    4    3    6   1   0    7   2   7
Rang     : 10    9    8    7    6   5   4    3   2   1
Produits :  0   72   32   21   36   5   0   21   4   7
Somme    =  0 + 72 + 32 + 21 + 36 + 5 + 0 + 21 + 4 + 7
         = 198
Division par 11 : quotient = 18 ; reste = 0
```

Pour le numéro EAN, l'ISBN est d'abord débarrassé de son chiffre de contrôle, 7 dans l'exemple précédent. Il est complété à gauche par 978. Pour l'exemple précédent, l'EAN devient : 978-0-8436-1072-? où le point d'interrogation remplace provisoirement le chiffre de contrôle du nouvel identifiant. Pour calculer la nouvelle valeur du chiffre de contrôle, chaque chiffre de l'EAN est affecté d'un coefficient de pondération alternativement égal à 1 ou à 3 en commençant par la gauche. La somme des produits des chiffres de l'EAN par leur coefficient est divisée par 10 et le reste obtenu est retranché à 10. Le résultat de la soustraction est le nouveau contrôle. Reprenons l'exemple précédent. Il vient :

```
EAN  :     9    7    8    0    8    4    3    6    1    0    7    2
Coef :     1    3    1    3    1    3    1    3    1    3    1    3
Produit :  9   21    8    0    8   12    3   18    1    0    7    6
Somme   =  9 + 21  + 8  + 0  + 8 + 12  + 3 + 18  + 1  + 0  + 7  + 6
        = 93
Division par 10 : quotient = 9 ; reste = 3
Contrôle        : 10 - reste = 10 - 3  = 7
EAN = 978-0-8436-1072-7
```

Exercice 8 : Vérifier des EAN

1. Les EAN suivants sont-ils valides ?

- *978-2-212-12136-0*
- *978-2-212-11026-5*
- *978-2-7440-1508-3*

2. Écrire les algorithmes de vérification des ISBN et EAN.

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a proposé des activités autour du problème de l'édition d'un nombre. Éditer un nombre, c'est déterminer sa représentation dans une base de numération donnée. Nous avons successivement étudié le calcul du nombre de chiffres d'un entier, son édition et le problème réciproque. Nous avons ensuite résolu le problème de la représentation d'un entier en chiffres romains. Les solutions au problème réciproque, qui consiste à transformer un nombre exprimé en chiffres romains en un entier, sont inspirées des solutions exposées dans [ARS80]. Les deux dernières sections ont étudié des algorithmes de vérifications, utilisés en informatique de gestion pour contrôler la cohérence de la saisie des identifiants d'entreprises ou de livres.

© ENI Editions - All rigths reserved

Bibliographie

[ARS80] Jacques ARSAC : *Premières leçons de programmation* ; CEDIC/NATHAN, 1980.

© ENI Editions - All rigths reserved

Introduction

Un système de traitement informatique des données ne peut pas toujours disposer, dans des variables définies, de toutes les entités sur lesquelles il intervient. Un ordinateur, par exemple, ne peut pas toujours placer toutes les données en mémoire centrale. C'est le cas lorsque le volume de données est important ou encore lorsqu'il est nécessaire d'assurer la persistance des informations qu'elles représentent. Des *unités périphériques externes d'enregistrement* sont alors utilisées pour enregistrer et récupérer les données. Ce chapitre étudie une forme particulière d'organisation et de structuration permettant la persistance des données impliquées dans un traitement algorithmique : les *fichiers*. Le prétexte est que la forme que revêt cette organisation impose des méthodes et des opérations applicables spécifiques.

La deuxième section donne quelques notions rapidement introduites sur l'organisation des supports externes d'information d'un système informatique et sur les modes d'accès à ces informations. La troisième section décrit un fichier à organisation séquentielle et les traitements de lecture et d'écriture dans un tel fichier. La section suivante donne quelques éléments sur les fichiers à organisation directe et accès sélectif.

© ENI Editions - All rigths reserved

Notions élémentaires

Cette section introduit rapidement quelques notions élémentaires sur l'organisation et l'accès aux informations enregistrées dans un fichier informatique. Elle est divisée en trois parties. La première est une introduction qui précise quelques éléments du vocabulaire du domaine. La deuxième partie présente les organisations usuelles. La troisième partie montre comment on définit une association, entre un fichier enregistré sur un support externe et une structure algorithmique de données, qui permettra de décrire les opérations d'accès aux informations enregistrées.

1. Fichiers et articles

On considère un ensemble d'entités particulières, comme les clients d'une entreprise, les nombres premiers inférieurs à une limite donnée, une collection de CD audio... Chacune des entités dont il s'agit, un client, un CD... est caractérisée par une collection d'informations. Ainsi, par exemple, un CD sera caractérisé par un titre, des interprètes, des titres de chansons s'il s'agit d'un CD de musique de variété... Nous avons vu au chapitre Structures élémentaires qu'une telle caractéristique est habituellement appelée un attribut. C'est l'unité d'information signifiante pour une entité. Dans le contexte de ce chapitre, on appelle *article logique* une telle collection d'attributs.

Définition

On appelle article logique *une collection d'attributs relatifs à une entité particulière.*

Pour organiser la persistance d'un ensemble d'articles logiques ayant des caractéristiques en commun, on les regroupe en *fichier*. Lorsque ce fichier est enregistré sur un support externe, souvent un support magnétique ou optique, on parle de *fichier physique* et un article est alors appelé un *article physique* ou *enregistrement*.

Définition

On appelle fichier logique *une collection d'articles logiques et* fichier physique *la copie du fichier logique enregistrée sur un support externe assurant sa persistance.*

On appelle article physique *ou* enregistrement *la copie de l'article logique dans le fichier physique.*

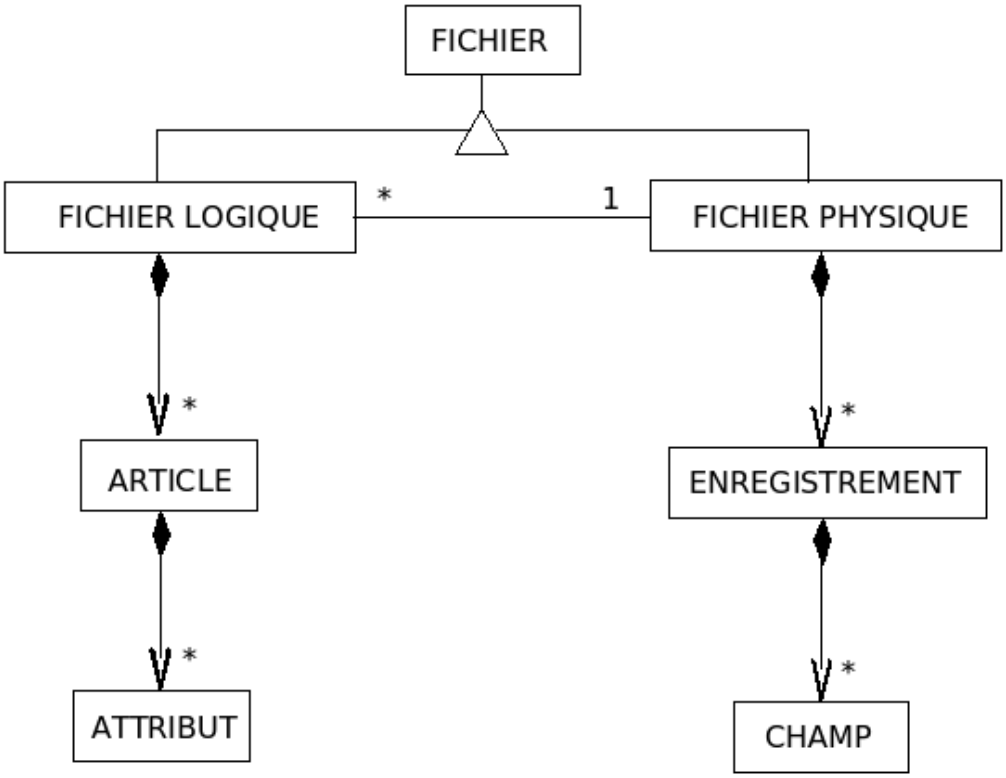
Lorsque l'article physique est en cause, on ne parle pas d'attribut pour désigner ses composants, mais de *champs* ou encore de *rubriques*.

Définition

Un attribut d'un enregistrement est appelé une rubrique *ou encore un* champ *de cet enregistrement.*

Finalement, un fichier logique est composé d'articles, eux-mêmes composés d'attributs. Un fichier physique, enregistré sur un support externe, est composé d'enregistrements, eux-mêmes composés de champs. La figure ci-dessous représente ces associations.

© ENI Editions - All rigths reserved



Ce schéma conceptuel précise que la famille des fichiers se décline en deux classes : les fichiers physiques et les fichiers logiques. Un fichier logique, par exemple, est composé d'aucun, d'un ou de plusieurs articles logiques. Le fichier peut donc être vide. Chaque article logique est composé d'aucun, d'un ou de plusieurs attributs. Ainsi, un article dans un fichier peut aussi être vide. La partie du schéma qui concerne un fichier physique se lit de la même façon.

Les opérations applicables à un fichier sont particulières du fait de l'enregistrement des fichiers physiques sur un support externe persistant. La notion de variable n'a plus de sens et les axiomes étudiés au chapitre Programmes directs ne sont plus adaptés. Une opération qui met en jeu un ensemble d'enregistrements est appelée une *transaction*.

Définition

On appelle transaction *une opération qui concerne un ensemble d'enregistrements.*

Ainsi considérons, par exemple, un fichier clients qui rassemble les renseignements associés aux clients d'une entreprise. La liste suivante énumère quelques exemples de transactions :

- déterminer la liste des clients qui n'ont rien commandé depuis 6 mois ;
- vieillir tous les clients de 1 an ;
- déterminer les clients qui habitent la région Centre.

Ce sont les fichiers logiques et leurs articles qui sont concernés par l'algorithmique. Les opérations sur les fichiers physiques et leurs enregistrements sont du domaine de la programmation. Évidemment, il existe un lien fort entre les deux familles de fichiers. On ne peut pas totalement s'abstraire de la signification des opérations sur un fichier logique pour un fichier physique. Dans les paragraphes suivants, on s'intéresse aux organisations des fichiers physiques sur leur support pour en déduire les opérations de base applicables à un fichier logique. L'adaptation aux fichiers physiques reste cependant du ressort de la programmation, ce qui dépasse les objectifs de ce livre.

2. Organisation et accès aux fichiers

Les fichiers physiques se répartissent en deux classes :

1. Les fichiers dans lesquels les informations sont enregistrées sous la forme d'un flot de caractères, au sens algorithmique du terme, comme cela a déjà été défini au chapitre Structures élémentaires : on les appelle des *fichiers texte* ;

© ENI Editions - All rigths reserved

2. Ceux dans lesquels les informations sont enregistrées sous la forme de données binaires : on les appelle des *fichiers binaires*.

Dans un fichier texte, toute information est enregistrée sous forme textuelle directement éditable, par exemple sur un écran informatique. Ainsi, le nombre mille deux cent soixante-quinze exprimé en base dix occupera 5 caractères sur le support, éventuellement augmentés des caractères de gestion propres au système informatique utilisé : les quatre caractères représentant ses chiffres '1', '2', '7', '5' et un caractère pour son signe. Par conséquent, l'espace de stockage nécessaire est directement fonction de la valeur du nombre. Dans un fichier binaire, au contraire, l'espace nécessaire reste constant. Il est indépendant de la valeur du nombre, dans les limites de la représentation du type auquel il appartient. Cependant, quelle que soit la classe du fichier, il est enregistré sur un support informatique dont on distingue deux catégories :

- Les *supports séquentiels* reçoivent les informations enregistrées *les unes à la suite des autres*. L'accès à une information nécessite l'accès préalable à toutes les informations qui la précèdent sur le support. C'est le cas, par exemple, de la bande magnétique ;
- Les *supports adressables* sont structurés en surfaces d'enregistrement individuellement adressables. Les informations qui y sont enregistrées peuvent être retrouvées directement grâce à leur adresse.

Selon l'organisation et les impératifs du traitement à réaliser, on utilise deux techniques d'accès aux enregistrements d'un fichier :

- L'*accès séquentiel* accède aux enregistrements du fichier dans l'ordre des enregistrements sur le support ;
- L'*accès sélectif* accède à un enregistrement particulier sans passer par les enregistrements intermédiaires. Cet accès n'est possible que sur les supports adressables.

L'organisation d'un fichier définit la manière dont les enregistrements du fichier sont disposés sur le support. On distingue trois organisations principales :

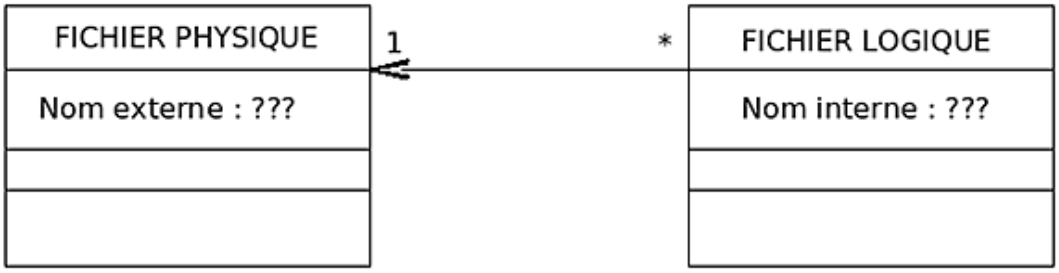
- L'*organisation séquentielle*, qui permet l'accès séquentiel ;
- L'*organisation directe* conçue pour l'accès sélectif ;
- L'*organisation séquentielle indexée* qui permet à la fois l'accès séquentiel et l'accès sélectif.

Cette introduction aux traitements des fichiers s'intéresse à l'organisation et à l'accès séquentiel ainsi qu'à quelques notions sur l'accès sélectif aux enregistrements de fichiers à organisation directe. Autrement dit, nous n'étudions, dans ce chapitre, que les méthodes algorithmiques relatives au fichier logique dont les enregistrements du modèle physique sont organisés séquentiellement ou directement.

3. Association d'un fichier physique à un programme

Un fichier physique F est identifié de manière unique par un *nom externe* qui lui est attaché. Le nom externe d'un fichier physique est le nom sous lequel il est connu du système d'exploitation ou, plus précisément, du système de gestion de fichiers (SGF) de l'hôte qui l'héberge. Ainsi, par exemple, le fichier qui contient le texte que vous êtes en train de lire s'appelle `fichiers.tex`. Son nom externe complet est : `/home/mon-login/livres/algorithmique/ch-10/fichiers.tex`

Il peut exister plusieurs programmes de traitement qui accèdent aux informations du fichier F. De même, un unique programme peut accéder au même fichier physique pour réaliser plusieurs transactions distinctes. Dans l'environnement de chacun d'eux, F, le fichier physique, est associé d'une façon unique à un fichier logique dont F est le modèle. La figure ci-dessous représente conceptuellement cette association.



© ENI Editions - All rigths reserved

Le fichier logique, celui de l'algorithmique, est identifié par un nom interne, généralement sans rapport avec le nom externe du fichier physique.

Afin de pouvoir opérer sur le fichier physique, un programme doit s'assurer le contrôle de ce fichier. Il utilise pour cela une opération d'*ouverture* du fichier externe. Cette opération lui permet :

- d'établir une correspondance entre le nom externe et un nom interne pour désigner le fichier logique ;
- de préciser la nature des opérations qui seront réalisées sur le fichier : lecture, écriture...
- d'assurer au programme le contrôle exclusif du fichier physique.

Ainsi, un même fichier physique ne possède qu'un et un seul nom externe. Cependant, plusieurs noms internes peuvent lui associer différents fichiers logiques. L'opération d'ouverture du fichier physique réalise l'une de ces associations. Un même fichier physique peut être associé à plusieurs fichiers logiques, mais un même fichier logique n'est associé qu'à un fichier physique unique. C'est ce qu'expriment les multiplicités. Chaque élément de l'ensemble OUVRIR[FICHIER_PHYSIQUE, FICHIER_LOGIQUE], autrement dit, chaque couple constitué d'un fichier logique et du fichier physique qui lui correspond, est obtenu par une opération d'ouverture spécifique.

Dans la suite, **ouvrir**(F, mode) notera l'opération d'ouverture qui réalise les actions décrites ci-dessus, permettant ainsi de traiter le fichier de nom externe F. Lorsque l'ouverture réussit, le nom externe F est associé à un fichier logique par le système de gestion de fichiers du système d'exploitation et la fonction **descripteur** retourne en résultat le nom interne du fichier logique. Cette correspondance permet d'accéder au fichier lors des opérations ultérieures. Cette fonction s'utilise comme ceci :

```
...
variable
    nom_interne : NOM_INTERNE # déclare un nom de fichier logique.
    nom_externe : NOM_EXTERNE # déclare le nom externe dans le SGF.
    mode        : MODE        # Nature des opérations à appliquer :
                              # LECTURE, ÉCRITURE, ...
...
initialisation
    initialiser nom_externe et mode
...
# Ouverture du fichier et initialisation du nom interne.
ouvrir(nom_externe, mode)
nom_interne <- descripteur(nom_externe)
...
```

Les types de données **NOM_EXTERNE** et **NOM_INTERNE** ne sont pas précisés pour l'instant. On peut anticiper en remarquant qu'un nom externe est le nom complet d'un fichier. C'est donc une chaîne de caractères. Le mode d'ouverture précise la nature des transactions valides sur ce fichier logique. Cependant, ce mode est contraint par l'organisation du fichier physique et le mode d'accès aux informations. Ainsi, par exemple, lorsque le support physique est une bande magnétique, on ne peut pas ouvrir le fichier en lecture et en écriture. Autrement dit, un même fichier physique ne peut être simultanément un fichier de sortie et un fichier d'entrée. Mais ce sont là des considérations relatives aux accès physiques. Nous en reparlerons plus bas.

MODE est un type structuré :

```
type
    MODE
structure
    défini par énumération {LECTURE, ÉCRITURE, LECTURE_BINAIRE,
    ÉCRITURE_BINAIRE}
fin MODE
```

La modélisation précédente montre bien l'association d'un nom interne à un nom externe, mais elle n'exprime pas toute la réalité du fonctionnement d'un SGF. Ce comportement dépend de l'organisation et du mode d'accès au fichier. La section suivante étudie plus en détail l'organisation et l'accès séquentiel.

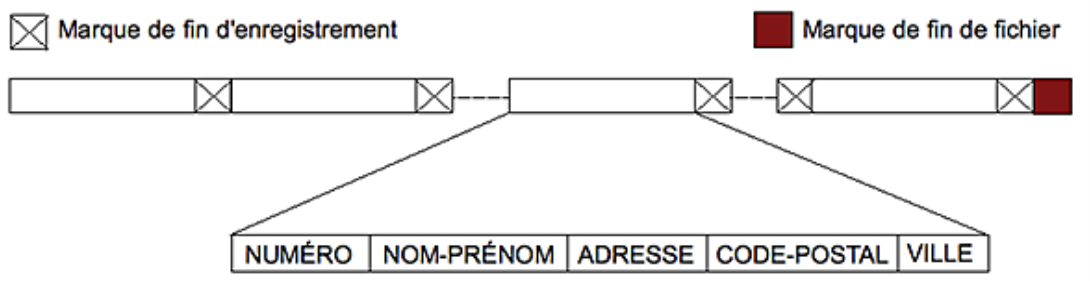
© ENI Editions - All rigths reserved

Organisation séquentielle

Cette section présente les algorithmes élémentaires permettant de définir les opérations de base sur un fichier logique dont le modèle est un fichier physique à organisation séquentielle. La première partie est une introduction qui précise le contexte. La deuxième partie montre comment parcourir un fichier pour lire les informations qu'il contient. La partie suivante en fait autant pour l'écriture dans le fichier. La dernière partie montre comment le mettre à jour.

1. Introduction

Un fichier physique à organisation séquentielle est constitué d'une suite d'enregistrements placés consécutivement sur un support externe. Ils sont tels que, pour accéder à un enregistrement donné e, il est nécessaire d'accéder d'abord et successivement à tous les enregistrements qui précèdent e sur le support. La figure suivante illustre ce type d'organisation.



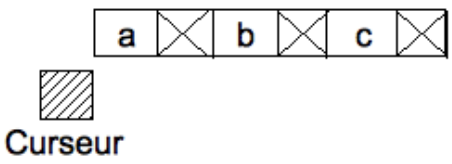
Les enregistrements d'un fichier sont habituellement composés. Sur la figure ci-dessus, chaque enregistrement est fait de cinq champs : NUMÉRO, NOM-PRÉNOM, ADRESSE, CODE-POSTAL, VILLE. On peut donc définir un article composant le fichier logique associé après ouverture. Ainsi, par exemple pour le fichier clients de la figure précédente, le type correspondant pourrait être :

```
type
    CLIENT
structure
    numéro     : ENTIER # Identifiant du client.
    identité   : IDENTITÉ
    adresse    : CHAÎNE
    codePostal : CODE_POSTAL
    ville      : CHAÎNE
fin CLIENT
```

Les types **IDENTITÉ** et **CODE_POSTAL** ont déjà été définis au chapitre Structures élémentaires. Dans ce qui précède, c'est l'article qui est l'unité élémentaire d'accès à un fichier logique.

Un fichier vide, qui ne contient aucun enregistrement, mais qui existe en tant que tel sur le support externe est un fichier réduit à la marque de fin de fichier.

Un fichier à organisation séquentielle peut être enregistré sur un support à accès séquentiel ou adressable. L'opération d'ouverture d'un fichier physique lui associe un nom interne utilisé pour les accès au fichier selon le mode précisé. L'effet de l'opération d'ouverture est de définir un curseur d'accès aux articles. Pour toute opération de lecture, un déplacement relatif préalable de ce curseur permet de réaliser l'opération. La figure ci-dessous représente la situation immédiatement après la réussite de l'opération d'ouverture.



Cette figure représente trois articles a, b et c, les marques de fin d'enregistrement et la position du curseur avant le premier article du fichier logique, c'est-à-dire en position 0. On suppose donc que la position du premier article, s'il existe, est 1. Lorsque le fichier est vide, son seul enregistrement est la marque de fin de fichier. Dans ce cas, le curseur est placé juste avant cette marque.

Pour opérer sur un fichier, la commande **ouvrir** crée le fichier logique et l'associe au fichier physique dont elle reçoit le nom externe en paramètre. C'est elle qui crée aussi le curseur d'accès aux articles du fichier. Cette commande est spécifiée par l'algorithme ci-dessous.

© ENI Editions - All rigths reserved

*Algorithme 1 : Spécifications de la commande **ouvrir***

```
Algorithme ouvrir
    # Créer un fichier logique associé à fichier dans le mode m.
Entrée
    fichier : NOM_EXTERNE # Le fichier à ouvrir.
    m : MODE # Nature des transactions à réaliser sur le fichier.
précondition
    # Pour une ouverture en LECTURE, le fichier existe dans le SGF.
    m = 'LECTURE' => existe(fichier)
    # Il ne doit pas être déjà ouvert en ÉCRITURE.
    non est_en_écriture(fichier)
    # Pour une ouverture en ÉCRITURE, il ne doit pas être déjà
    # ouvert.
    m = 'ÉCRITURE' => non est_en_lecture(fichier) et
                      non est_en_écriture(fichier)
postcondition
    # Le fichier logique est dans l'état OUVERT.
    est_ouvert(descripteur(fichier))
    # Le fichier physique est verrouillé dans le mode m.
    m = 'LECTURE'  => est_en_lecture(fichier)
    m = 'ÉCRITURE' => est_en_écriture(fichier)
    # Les paramètres n'ont pas été modifiés.
    ancien(m) = m
    ancien(fichier) = fichier
fin ouvrir
```

Lorsque le fichier est ouvert, il est verrouillé dans le mode choisi. Les prédicats **est_en_lecture** et **est_en_écriture** permettent d'interroger le système logiciel pour déterminer ce mode. Les raisons pour lesquelles l'ouverture d'un fichier en écriture suppose que le fichier n'est pas déjà ouvert seront exposées plus bas. Cette spécification est donc claire, sauf peut-être pour ce qui concerne la fonction **descripteur**. Cette requête rend le nom interne du fichier dont elle reçoit en paramètre le nom externe. Comme un même fichier peut être ouvert plusieurs fois, comme l'a montré la section précédente, **descripteur** rend le dernier nom interne associé au fichier, donc celui obtenu lors de la dernière ouverture.

En fin de traitement, les ressources mobilisées par l'ouverture sont libérées en fermant le fichier. L'opération correspondante est la procédure **fermer**. Elle réalise les opérations suivantes :

- elle complète le fichier physique par une marque de fin de fichier lorsque le traitement réalisé était une création, donc lorsque le fichier était ouvert en écriture ;
- elle libère toutes les ressources mobilisées pour le traitement du fichier et, en particulier, elle libère l'accès au fichier physique qui devient disponible pour les autres traitements. Le nom interne particulier utilisé pour les traitements précédents n'a plus de signification et peut, par exemple, être réutilisé par le système d'exploitation pour un autre fichier.

La commande **fermer** termine donc l'association entre un fichier physique et un fichier logique. Les spécifications de l'algorithme **fermer** sont les suivantes :

*Algorithme 2 : Spécifications de l'opération **fermer***

```
Algorithme fermer
    # Ferme fichier et libère les ressources.
Entrée
    fichier : NOM_INTERNE # Le fichier logique à fermer.
précondition
    # Le fichier logique est ouvert.
    est_ouvert(fichier)
postcondition
    # Le fichier est fermé et déverrouillé par le SGF.
    non est_ouvert(fichier)
    # Le paramètre n'est pas modifié.
    ancien(fichier) = fichier
fin fermer
```


© ENI Editions - All rigths reserved

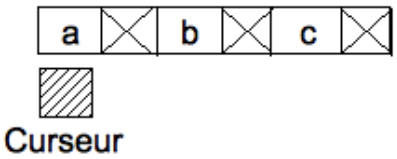
Une opération quelconque sur un fichier, comme **ouvrir**, **fermer**… induit des accès au périphérique de stockage et au support externe. Ces accès ne sont pas toujours possibles, par exemple quand le périphérique est en panne. Dans ce cas, une erreur se produit et le programme informatique doit assurer les traitements de toutes les situations dans lesquelles surviennent ces cas exceptionnels. Dans la suite, nous ignorons des erreurs, pannes, défauts… qui résultent du fonctionnement de la machine physique et nous supposons qu'elles sont systématiquement reportées vers les modules logiciels qui utilisent les services des algorithmes que nous écrivons. Cette simplification n'est pas satisfaisante quand on étudie les algorithmes de solutions à des problèmes industriels réels, mais nous nous en contenterons dans cette initiation.

Nous sommes prêts à étudier comment lire dans un fichier à organisation et accès séquentiels.

2. Traitement d'un fichier séquentiel en lecture

Cette section présente des algorithmes élémentaires permettant de définir les opérations de base sur un fichier logique dont le modèle est un fichier physique à organisation et accès séquentiels.

Pour une lecture du fichier, la procédure **premier** a pour effet de placer le curseur de lecture immédiatement sous le premier article. C'est la marque de fin de fichier lorsque le fichier est vide. La fonction **position** rend alors 1. La figure précédente a illustré la situation immédiatement après ouverture du fichier. La figure suivante l'illustre après l'appel à la procédure **premier**.



L'algorithme qui précise les spécifications de cette procédure est le suivant :

*Algorithme 3 : Spécifications de **premier***

```
Algorithme premier, début
    # Place le curseur de lecture sous le premier article.
Entrée
    fichier : NOM_INTERNE
précondition
    # Le fichier est ouvert en lecture.
    est_en_lecture(fichier)
postcondition
    # Le curseur est placé en première position.
    position(fichier) = 1
    # Fichier n'est pas modifié.
    ancien(fichier) = fichier
fin premier
```

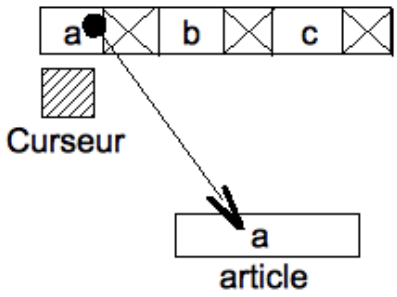
Cet algorithme définit aussi **début** comme un alias de **premier** pour des raisons de commodité.

L'effet de la fonction **lire** est de renvoyer l'article qui se trouve en face du curseur. Des appels successifs à **lire** retournent systématiquement la même valeur du même article. Autrement dit, **lire** ne modifie pas la position du curseur. Considérons, par exemple, les instructions suivantes :

```
...
fichier : NOM_EXTERNE # Le fichier à traiter.
f       : NOM_INTERNE # L'identifiant interne du fichier.
article : ARTICLE # Un article du fichier.
...
ouvrir(fichier, 'LECTURE') # Ouvre le fichier en lecture.
f <- descripteur(fichier)  # Identifiant attribué à l'ouverture.
premier(f)                 # Placer le curseur sous le premier article.
article ← lire(f)          # Récupérer l'article sous le curseur.
```

La variable `article` contient alors la valeur de l'article en face du curseur. Cet article peut être la marque de fin de fichier si le fichier est vide. Le dessin suivant représente cette situation après l'appel à **premier**, suivi de l'appel à la fonction **lire**.

© ENI Editions - All rigths reserved



lire réalise donc une copie de l'article sous lequel est placé le curseur.

*Algorithme 4 : Spécifications de la fonction **lire***

```
Algorithme lire
    # L'article de fichier à la position du curseur.
Entrée
    fichier : NOM_INTERNE # Le fichier à lire.
Résultat : ARTICLE
précondition
    # Fichier est ouvert en lecture.
    est_en_lecture(fichier)
    # La position du curseur est au moins celle du premier article.
    position(fichier) > 0
postcondition
    Résultat = (l'article à la position du curseur)
fin lire
```

Cet algorithme est une fonction. Il est donc inutile de préciser que la position du curseur n'est pas modifiée. Ainsi, une nouvelle lecture du fichier rend le même article. Pour accéder à l'article suivant et obtenir sa valeur, il est nécessaire de commencer par avancer le curseur. Soit **avancer** la procédure qui réalise cette modification. Ses spécifications sont :

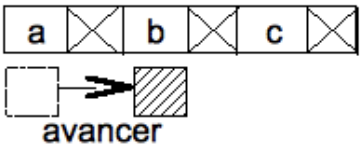
*Algorithme 5 : Spécifications de **avancer***

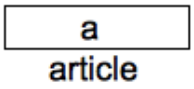
```
Algorithme avancer
    # Déplacer le curseur sous l'article suivant.
Entrée
    fichier : NOM_INTERNE
précondition
    # Le fichier est ouvert.
    est_ouvert(fichier)
    # En lecture, le curseur est avant la marque de fin de fichier.
    est_en_lecture(fichier) => non estAprès(fichier)
postcondition
    # Le curseur n'avance pas après la fin de fichier.
    finDeFichier(ancien(fichier))     =>
                position(fichier) = ancien(position(fichier))
    # Le curseur avance d'une position si non fin de fichier.
    non finDeFichier(ancien(fichier)) =>
                position(fichier) = ancien(position(fichier)) + 1
fin avancer
```

Le prédicat **estOuvert** n'est qu'un raccourci pour la disjonction **est_en_lecture() ou est_en_écriture()**.

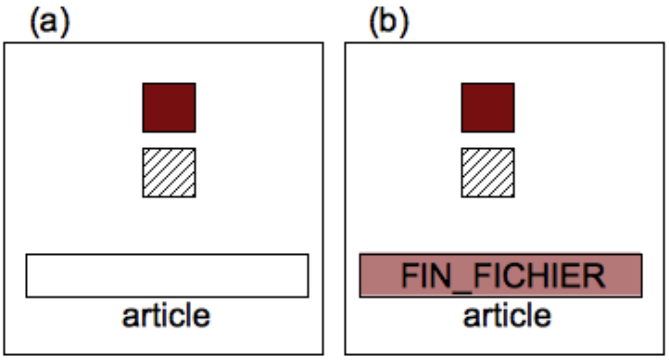
Le prédicat **estAprès** rend VRAI si et seulement si le curseur est passé après la position de la marque de fin de fichier. Ainsi, cette commande n'a aucun effet lorsque le curseur est passé sous cette marque. La figure ci-dessous illustre la situation après exécution de la commande **avancer**.

© ENI Editions - All rigths reserved





Lorsque la fin de fichier est rencontrée, la variable `article` contient FIN_FICHIER. C'est une constante manifeste dont la valeur est celle du caractère spécial qui marque la fin du fichier. Pour reconnaître cette situation, on dispose d'une fonction booléenne **finDeFichier** qui retourne FAUX à l'ouverture du fichier et tant qu'une opération de lecture n'a pas rencontré ou dépassé la marque de fin de fichier. Elle rend VRAI lorsque le curseur est positionné sous cette marque et que la fonction de lecture a été appelée dans cette position. Ainsi, même si le fichier est vide, cette fonction ne renvoie VRAI qu'après une première lecture ou après avoir avancé le curseur au-delà de la marque. La figure suivante illustre cette situation.



La partie (a) du dessin illustre la situation après ouverture d'un fichier vide et appel à la procédure **premier**. Le curseur est placé alors sous la marque de fin de fichier. La valeur de la variable `article` n'est pas encore significative et le prédicat **finDeFichier** rendra FAUX s'il est appelé dans cette position du curseur. La partie (b) montre la situation après lecture depuis la situation illustrée par (a). Cette fois, la variable `article` a pris la valeur FIN_FICHIER et la fonction **finDeFichier** rendra VRAI au prochain appel.

L'algorithme des spécifications de **finDeFichier** est le suivant :

*Algorithme 6 : Spécifications de **finDeFichier***

```
Algorithme finDeFichier
    # La marque de fin de fichier a-t-elle été rencontrée ?
Entrée
    fichier : NOM_INTERNE
Résultat : BOOLÉEN
précondition
    # Le fichier est ouvert en lecture.
    est_en_lecture(fichier)
postcondition
    position(fichier) = 0 => Résultat = FAUX # Après ouverture.
    position(fichier) > 0 => Résultat =
        (
            estAprès(fichier)
          ou sinon
            (lire(fichier) = FIN_FICHIER)
        )
fin finDeFichier
```

On exprime ainsi que la fin de fichier est atteinte si et seulement si une lecture rend la marque de fin de fichier ou si le curseur est placé après cette marque.

Il devient possible de commencer à résoudre quelques exercices.

© ENI Editions - All rigths reserved

Exercice 1 : Quelques spécifications complémentaires

*Soit **avancer**(fichier, p) qui avance le curseur d'un nombre de positions égal à p.*

1. Donner les spécifications de cette nouvelle procédure.

2. Vérifier que ces spécifications sont cohérentes avec le fonctionnement décrit précédemment.

En particulier, comment se comporte cette procédure sur un fichier vide ? Plus généralement, comment se comporte-t-elle lorsque le nombre p est tel que l'avance du curseur lui fait dépasser la position de la marque de fin de fichier ?

Quelques prédicats utilisés n'ont pas encore été définis.

*3. Écrire les spécifications des prédicats **est_en_lecture** et **est_en_écriture**.*

*4. Utiliser les spécifications obtenues à la question précédente pour spécifier **est_ouvert**.*

*5. Utiliser ces spécifications pour définir une fonction **mode** qui rend le mode dans lequel a été ouvert un fichier.*

3. Parcours d'un fichier

Parcourir un fichier séquentiel est une opération fréquente. L'exercice suivant donne des indications pour réaliser cette opération.

Exercice résolu 1 : Parcours d'un fichier à organisation séquentielle

Un fichier de nom externe F doit être parcouru pour faire subir à chacun de ses enregistrements un traitement qui n'est pas précisé.

1. Écrire l'algorithme général de parcours du fichier.

2. Comment repositionner le curseur au début du fichier pour un nouveau parcours ?

On recherche un article particulier parmi les n articles du fichier. Pour le trouver, il est nécessaire de parcourir le fichier depuis le premier article, jusqu'à rencontrer l'article cherché ou jusqu'à la marque de fin de fichier si cet article est absent du fichier.

3. Écrire l'algorithme de recherche d'un article dans un fichier séquentiel.

Solution

Voici, à titre d'exemple, le squelette d'un algorithme qui répond partiellement à la première question. Il n'est pas difficile d'en faire un algorithme complet et cette tâche est laissée en exercice.

```
...
# Ouvrir le fichier F en lecture et récupérer le nom interne obtenu.
ouvrir(F, 'LECTURE')
fichier <- descripteur(F)
# Positionner le curseur sous le premier article.
premier(fichier) # ou début(fichier).
# Lire le premier article.
article <- lire(fichier)
jusqu'à
    finDeFichier(fichier)
répéter
    Réaliser les traitements sur article
    # Placer le curseur sous l'article suivant.
    avancer(fichier)
    # Lire l'article suivant.
    article <- lire(fichier)
fin répéter
fermer(fichier)
...
```

Cette définition est évidemment incomplète, mais il y manque aussi l'invariant et le variant de contrôle de l'itération. L'invariant n'est pas difficile à écrire. Il n'en est pas de même du variant de contrôle qui peut être ignoré.

Remarquer aussi que l'appel au prédicat **finDeFichier** est équivalent au prédicat **lire**(fichier) = FIN_FICHIER.

Pour repositionner le curseur sous le premier article, il suffit de fermer puis de rouvrir le même fichier :

© ENI Editions - All rigths reserved

```
...
fermer(fichier)
# Ouvrir le fichier F en lecture et récupérer le nom interne obtenu.
ouvrir(F, 'LECTURE')
fichier <- descripteur(F)
# Positionner le curseur sous le premier article.
premier(fichier)
...
```

Il est possible de définir pour cela une procédure qui encapsule ces instructions :

Algorithme 7 : Replacer le curseur sous le premier article

```
Algorithme début2
    # Reculer le curseur sous le premier article.
Entrée
    fichier : NOM_INTERNE
précondition
    est_en_lecture(fichier)
réalisation
    fermer(fichier)
    # Ouvrir le fichier F en lecture et récupérer le nom interne
    # obtenu.
    ouvrir(F, 'LECTURE')
    fichier <- descripteur(F)
    # Positionner le curseur sous le premier article.
    premier(fichier)
postcondition
    ancien(fichier) = fichier
    est_en_lecture(fichier)
    position(fichier) = 1
fin début2
```

La première clause de la postcondition exprime que le nom interne fichier n'est pas modifié. En situation réelle, ce ne sera pas le cas. Nous ne pouvons pas supposer que le SGF rendra le même nom interne puisque le nom précédent est libéré par l'opération de fermeture et que beaucoup d'événements peuvent se passer entre cette fermeture et la réouverture du même fichier. Cependant, les langages de programmation, comme C par exemple, fournissent une procédure **rewind** qui a exactement le comportement décrit par **début2** et nous pouvons en rester là.

La dernière question pose un problème moins immédiat. Soit **CLIENT** le type défini par la déclaration :

```
type CLIENT
    numéro : ENTIER
    nom    : CHAÎNE
fin CLIENT
```

Un client est donc une entité définie par deux attributs : un numéro entier, que nous supposerons positif et qui identifie le client ; un nom qui représente son identité.

Un fichier physique F à organisation et accès séquentiel contient jusqu'à 1000 clients enregistrés dans l'ordre croissant de leur numéro. Ces numéros ne sont pas nécessairement consécutifs. *On demande l'algorithme qui rend le nom d'un client dont on donne le numéro.*

*Algorithme 8 : Fonction **unClient***

```
Algorithme unClient
    # Le nom du client de numéro numéro ou ABSENT.
Entrée
    F      : NOM_EXTERNE # Fichier des clients.
    numéro : ENTIER      # Numéro du client à chercher.
Résultat : CHAÎNE        # Le nom cherché ou ABSENT.
précondition
    existe(F)           # Le fichier existe dans le SGF.
    non est_ouvert(F)   # Il n'est pas déjà verrouillé.
constante
```

© ENI Editions - All rigths reserved

```
    MAX_CLIENTS : ENTIER <- 1000
    ABSENT      : CHAÎNE <- CHAÎNE_VIDE
variable
    clients : NOM_INTERNE # Nom interne de F après ouverture.
    nom     : CHAÎNE      # Le nom du client cherché.
réalisation
    # Créer le fichier logique.
    ouvrir(F, 'LECTURE')
    clients <- descripteur(F)
    # Déterminer le nom du client de numéro numéro.
    Résultat <- identité(clients, numéro).nom
    # Libérer les ressources.
    fermer(fichier)
postcondition
    Résultat = nom du client de numéro ou sinon ABSENT
fin unClient
```

C'est la fonction **identité** qui réalise le parcours effectif du fichier.

*Algorithme 9 : Fonction **identité***

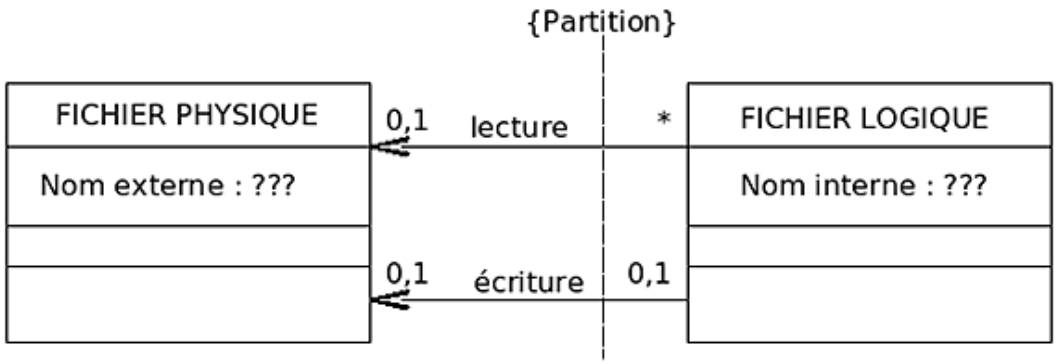
```
Algorithme identité
    # Le client associé à numéro.
Entrée
    clients : NOM_INTERNE # Le fichier à explorer.
    numéro  : ENTIER      # Numéro du client cherché.
    client  : CLIENT      # Le prochain client à observer.
Résultat : CLIENT
précondition
    est_en_lecture(clients)
    1 ≤ numéro ≤ MAX_CLIENTS
réalisation
    Résultat <- ABSENT      # Encore rien trouvé.
    premier(clients)        # Curseur sous le premier article.
    client <- lire(clients) # Lecture du premier client.
    tant que
        non finDeFichier(clients) et Résultat = ABSENT
        invariant
            ...
        variant de contrôle
            ...
    répéter
        si
            client.numéro = numéro
        alors
            # Trouvé.
            Résultat <- client
        sinon
            # Pas encore trouvé. Curseur sous l'article suivant.
            avancer(clients)
            # Lire le client suivant.
            client <- lire(clients)
        fin si
    fin répéter
postcondition
    Résultat = ABSENT ou sinon Résultat.numéro = numéro
fin identité
```

Écrire dans un fichier à organisation et accès séquentiel est une opération essentiellement différente de la lecture. La section suivante étudie cette opération.

4. Traitement en écriture

© ENI Editions - All rigths reserved

Comme pour la lecture d'un fichier à organisation et accès séquentiel, l'écriture suppose que le fichier a d'abord été ouvert dans le mode correspondant. C'est encore l'opération **ouvrir** qui assure la création du nom interne, récupéré ensuite par la fonction **descripteur**. Cependant, alors que la fermeture du fichier ouvert en lecture n'avait que pour effet de libérer les ressources mobilisées, notamment le nom interne, elle a cette fois un effet plus définitif : c'est cette opération qui appose la marque de fin de fichier après la marque de fin du dernier enregistrement. Ainsi, toute ouverture d'un tel fichier en écriture aura pour conséquence que sa fermeture marquera la fin du fichier à la position du curseur. Par conséquent, le simple fait d'ouvrir le fichier en écriture le crée initialement vide et, donc, si le fichier physique contenait des données avant l'ouverture en écriture, ces données sont perdues. En effet, la fermeture du fichier apposera la marque de fin de fichier et tout enregistrement placé après la marque de fin de fichier est « invisible » et donc perdu s'il existait avant l'ouverture. Une première conséquence importante est qu'il devient nécessaire d'exprimer qu'un fichier ouvert en lecture ne peut être ouvert en écriture et que, réciproquement, un fichier ouvert en écriture ne peut l'être ensuite et en même temps en lecture. Ainsi, le modèle conceptuel suivant précise mieux cette réalité :



La contrainte de partition, notée {Partition}, exprime qu'un fichier ne peut être ouvert simultanément en lecture et en écriture, mais qu'il l'est soit en lecture soit en écriture. Ainsi, un fichier peut être ouvert aucune, une ou plusieurs fois en lecture et alors, il ne peut être ouvert en écriture. De même, un fichier peut être ouvert au plus une fois en écriture et alors, il ne peut être ouvert en lecture.

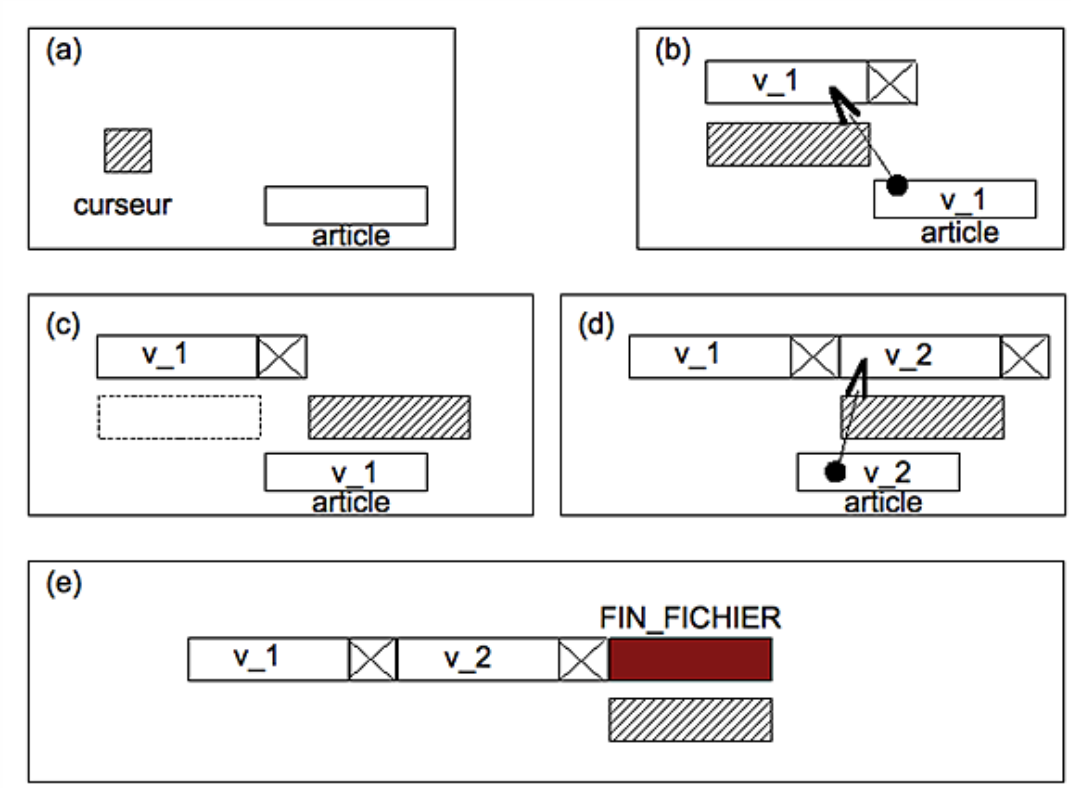
L'opération :

```
ouvrir(nom_externe, 'ÉCRITURE')
nom_interne <- descripteur(nom_externe)
```

crée le fichier de nom `nom_externe` sur le support, l'ouvre en écriture et retourne le nom interne pour accéder au fichier logique. L'opération **écrire**(nom_interne, valeur) écrit l'article désigné par la variable `valeur` dans la position du curseur et termine par l'apposition de la marque de fin d'enregistrement. L'opération **avancer**(nom_interne) avance alors le curseur après la marque de fin d'enregistrement qui vient d'être écrite et, enfin, l'opération **fermer** (nom_interne) appose la marque de fin de fichier et libère le nom interne.

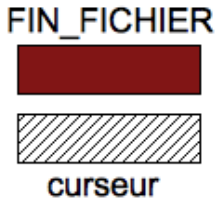
Considérons un état initial dans lequel on veut créer un fichier de nom externe F pour y enregistrer deux valeurs v_1 et v_2 de type **T**. La figure suivante illustre cinq états du fichier. La partie (a) montre l'état initial, après l'ouverture du fichier. La partie (b) représente la situation après l'écriture de la valeur v_1.

© ENI Editions - All rigths reserved



Pour écrire une nouvelle valeur v_2, on avance d'abord le curseur après la marque de fin d'enregistrement de l'écriture précédente. C'est ce qu'illustre la figure (c) ci-dessus qui montre la nouvelle situation après le déplacement du curseur. La figure (d) la montre après écriture de la valeur v_2 dans la nouvelle position du curseur. Enfin, la figure (e) montre l'effet de la fermeture du fichier après avance du curseur à la position qui suit la dernière écriture.

Lorsque les opérations d'ouverture et de fermeture se succèdent sans opération d'écriture intermédiaire, le fichier créé est un fichier vide, dans lequel ne figure que la marque de fin de fichier apposée par l'opération de fermeture. La figure suivante illustre cette situation.



Exercice résolu 2 : Création d'un fichier à organisation séquentielle

On considère un traitement général non précisé qui crée des valeurs v_i lors d'une itération. Ces valeurs doivent toutes être enregistrées dans un fichier à organisation séquentielle de nom externe F.

Écrire l'algorithme de création du fichier.

Solution

Le squelette d'un algorithme de création est le suivant :

```
...
variable
    F      : NOM_EXTERNE # Nom du fichier dans le SGF.
    f      : NOM_INTERNE # Nom du fichier logique après ouverture.
    mode   : MODE        # Mode (type) d'ouverture.
    valeur : ARTICLE     # Un article à écrire.
...
initialisation
    F <- (le nom externe du fichier à créer)
    mode <- 'ÉCRITURE'
```


© ENI Editions - All rigths reserved

```
    ouvrir(F, mode)
    f <- descripteur(F)
    valeur <- (calculer un article à enregistrer)
jusqu'à
    plus d'autre article à enregistrer
répéter
    écrire(f, valeur) # Enregistrer cette valeur et poser la marque
                      # de fin d'enregistrement.
    avancer(f)        # Position d'écriture suivante.
    valeur <- (calculer la valeur de l'article suivant)
fin répéter
fermer(f) # Poser la marque de fin de fichier et libérer les
          # ressources.
...
```

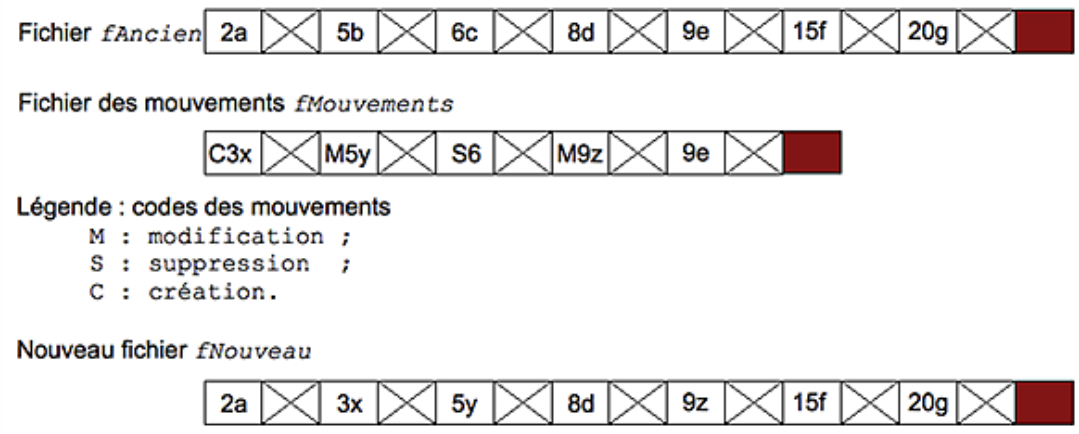
5. Mise à jour d'un fichier à organisation séquentielle

La mise à jour d'un fichier est l'opération qui consiste à modifier les informations qu'il contient pour :

- supprimer des articles dont les informations sont périmées ;
- créer de nouveaux articles ;
- modifier des articles existants.

Comme nous l'avons vu dans les sections précédentes, il n'est pas possible d'écrire dans un fichier à organisation séquentielle sans le créer. Par conséquent, toute écriture dans un tel fichier commence par détruire les informations qu'il contient. C'est le comportement décrit précédemment puisque la fermeture du fichier ouvert en écriture appose la marque de fin de fichier dans la position du curseur. Par conséquent, *la mise à jour d'un fichier à organisation et accès séquentiel s'effectue toujours en création !* Soit, par exemple, *fAncien* le nom externe du fichier à mettre à jour. Les opérations de mise à jour sont enregistrées dans un fichier de nom *fMouvements*. Ce fichier contient des enregistrements complétés d'un code qui précise, pour chacun d'eux, s'il s'agit de la modification d'un enregistrement existant dans *fAncien*, de la suppression d'un enregistrement existant ou de la création d'un nouvel enregistrement. Les enregistrements des fichiers *fAncien* et *fMouvements* sont utilisés pour créer un fichier *fNouveau* qui sera *fAncien*, mis à jour avec les informations de *fMouvements*. À l'issue de la mise à jour, *fAncien* est archivé ou détruit et *fNouveau* est renommé en *fAncien* pour le remplacer.

Considérons la figure ci-dessous. Elle représente les enregistrements des deux fichiers *fAncien* et *fMouvements*, ainsi que ceux du fichier *fNouveau* obtenu après mise à jour de *fAncien* à l'aide des enregistrements de *fMouvements*.



Cette figure représente le fichier à mettre à jour et le fichier des mouvements à réaliser. Ce dernier fichier rassemble des enregistrements composés de trois champs :

- un code de mouvement à réaliser appartenant à l'ensemble {M ; S ; C} ;
- une clé numérique ;

© ENI Editions - All rigths reserved

- la partie des données « utiles » de l'enregistrement, représentée ici par une lettre unique.

Ainsi, par exemple, le premier enregistrement `C3x` est la création de code `'C'`, d'un article de clé 3 dont les données sont représentées par la lettre `'x'`.

Exercice 2 : Mise à jour d'un fichier à organisation et accès séquentiel.

On suppose que les enregistrements des fichiers sont triés en ordre croissant des clés numériques. Ici, elle est représentée par le nombre entier en première position de chaque enregistrement de `fAncien`.

1. Écrire l'algorithme de mise à jour du fichier à partir du fichier des mouvements.

On s'intéresse à présent à la fusion de deux fichiers.

Deux fichiers à organisation séquentielle F1 et F2 sont constitués d'enregistrements ordonnés sur la valeur de l'un des champs K. On accède à la valeur de ce champ en utilisant la notation pointée. Ainsi, par exemple, après la lecture d'un article désigné par `ValeurLue`, on accède à la rubrique K de cet article en écrivant `ValeurLue.K`.

2. Écrire l'algorithme qui réalise la fusion des deux fichiers en un fichier unique.

Certaines opérations utiles n'ont pas encore été étudiées. Ainsi, par exemple, la procédure **dernier**, qui positionne le curseur sous le dernier article d'un fichier ouvert en lecture, n'a pas encore été définie.

Exercice 3 : Généralisation des spécifications

1. Établir la liste des opérations utiles et en donner les spécifications avec, éventuellement, les définitions correspondantes.

*La désignation des positions des articles utilise la fonction **position**. Au lieu de considérer que cette fonction retourne 1 lorsque le curseur est placé sous le premier article d'un fichier ouvert en lecture, on suppose que ce nombre est quelconque mais entier. Sa valeur est obtenue par la fonction **position_min**.*

2. Reprendre les spécifications des opérations et les algorithmes déjà étudiés avec ces nouvelles conventions.

© ENI Editions - All rigths reserved

L'organisation directe et l'accès sélectif

Dans certaines applications, chaque article contient un attribut particulier qui l'identifie d'une manière unique dans l'ensemble des articles qui composent un fichier. On appelle *clé* ou *identifiant* un tel attribut.

Définition

On appelle clé *d'un article d'un fichier un attribut qui identifie d'une manière unique cet article dans le fichier.*

Ainsi, deux articles distincts ont des clés différentes et, inversement, deux clés distinctes identifient deux articles distincts. Cette situation est essentiellement différente de celle qui fait d'un attribut K d'un fichier à organisation séquentielle une clé permettant d'ordonner les articles. En effet, dans ce cas, il pourrait exister plusieurs articles ayant la même valeur pour l'attribut K qui n'est donc pas une clé identifiante au sens de la définition précédente.

Lorsque le fichier est implanté sur un support adressable et que, de plus, une fonction permet d'établir une correspondance entre la clé d'un article et l'adresse de l'enregistrement associé à cet article, on peut accéder aux enregistrements d'un fichier *directement*, sans avoir à parcourir tous ceux qui le précèdent sur le support. Sur un support adressable, chaque emplacement est localisé par une *adresse*. Cependant, en algorithmique, on considère que chaque article est repéré par une *adresse relative* au début du fichier et non pas par rapport à un support physique programmable. La correspondance entre une adresse relative au début du fichier et l'adresse réelle sur le support est établie par les primitives d'accès au support. Ces primitives sont fournies par le système d'exploitation et son SGF et nous n'avons pas à nous en préoccuper. L'avantage d'une telle abstraction est qu'une réorganisation de la distribution des fichiers dans le système n'a pas de conséquence sur les algorithmes et les programmes d'accès aux données.

La suite de cette section ne traite que de deux formes de correspondances entre la clé et l'adresse d'un article. La première section étudie la technique de la *table d'accès* pour des fichiers stables. La section suivante étudie la méthode des *fonctions de répartition*.

1. Correspondance à l'aide d'une table d'accès

On considère un fichier de faible volume, comprenant peu d'articles et stable, c'est-à-dire évoluant peu. Ainsi, les mises à jour y sont peu fréquentes et les accès se font pour la plupart en lecture. Dans ce cas, la correspondance entre les clés et les adresses des enregistrements identifiés par ces clés est établie à l'aide d'une *table d'accès*. C'est un tableau ordonné selon la valeur des clés qu'il contient. Ce tableau est enregistré sur le support, avec le fichier qu'il permet d'adresser. En début de traitement sur le fichier, la table d'accès est chargée en mémoire centrale. Autrement dit, on constitue dans une variable de type tableau la table d'accès. Pour chaque accès à un enregistrement, on détermine d'abord son adresse en effectuant une recherche par dichotomie dans la table d'accès.

Exercice 4 : Organisation directe et table d'accès

On considère un fichier `clients.txt` *dont les enregistrements sont structurés par le type* ***CLIENT*** *défini au chapitre précédent. La clé identifiant un enregistrement est l'attribut numéro.*

Le fichier est d'abord considéré comme un fichier à accès séquentiel.

1. Écrire l'algorithme qui calcule la table d'accès et la trie en ordre croissant des clés.

L'adresse d'un article sera la position du curseur de lecture lorsqu'il est placé sous cet article. Dans le fichier, les enregistrements ne sont pas nécessairement enregistrés dans un ordre particulier des clés.

2. Écrire l'algorithme qui enregistre la table d'accès calculée dans un fichier séquentiel secondaire.

3. Écrire l'algorithme qui lit la table d'accès depuis le fichier secondaire et la reconstitue dans une variable tableau.

4. Écrire l'algorithme de parcours du fichier `clients.txt` *dans l'ordre croissant des clés.*

5. Écrire l'algorithme de recherche dichotomique d'une adresse à partir de sa clé.

2. Correspondance à l'aide d'une fonction de répartition

Dans la solution précédente, toute mise à jour du fichier impose une réorganisation complète de la table d'accès. C'est ce qui explique que cette méthode ne soit adaptée qu'à des fichiers stables, subissant peu de modifications. Une méthode plus efficace, mieux adaptée aux différents types de fichiers, utilise un *adressage associatif*, encore appelé adressage par *table de hachage* ou *adressage calculé*. Cette technique est bien adaptée aux recherches dans des éléments non ordonnés. Cette section présente une forme simplifiée d'adressage associatif sur un exemple élémentaire. L'adressage calculé est présenté dans l'exercice suivant.

Exercice 5 : Présentation de l'adressage calculé

© ENI Editions - All rigths reserved

Soit M l'ensemble des mots {bijou ; caillou ; chou ; genou ; hibou ; joujou ; pou}. Ce sont les mots de la langue française qui prennent un « x » terminal au pluriel. Soit H la fonction définie de M dans ℕ par :

$$H(m) = reste([\sum_{j=1}^{j=longueur(m)} code(item(m, j))], 10)$$

Cette expression signifie que, pour un mot `m` donné, on additionne les codes des caractères qui le composent et on calcule le reste de la division de cette somme par 10.

1. Créer le tableau H pour y enregistrer les mots de M. Chaque mot occupe la composante dont le numéro est la valeur renvoyée par la fonction H.

2. Comment retrouver, dans ce tableau, la position d'un mot donné ? Le temps de recherche dépend-il du nombre de mots de la table ?

3. Compléter la table en ajoutant les mots coucou, bignou, époux et ventou. Que remarquez-vous ? Comment éviter ce problème ?

La fonction `H` « condense » les clés de tous les mots sur dix valeurs différentes. Les collisions sont donc certaines et inévitables dès que le nombre de mots est supérieur à 10. En fait, il est impossible de définir une fonction de répartition qui réalise une bijection entre les ensembles considérés. Les notes bibliographiques donnent des références qui étudient d'une façon approfondie les techniques de hachage.

Soit `n` le nombre d'enregistrements d'un fichier. La fonction de répartition détermine, pour chacune des valeurs des `n` clés, l'adresse relative de l'enregistrement. Bien entendu, le système logiciel doit résoudre les collisions en proposant un mécanisme de gestion adapté. Ces techniques dépassent le cadre de cette initiation simple. Nous nous contentons de l'exercice suivant, qui évacue le problème en enregistrant les clés conduisant à une collision.

Exercice 6 : Fichier et fonction de répartition

On considère encore le fichier `clients.txt` de l'exercice précédent dont on suppose qu'il contiendra au plus n = 10000 enregistrements. La fonction `H` utilise, pour le calcul du reste, un dividende `d` choisi de façon que l'espace libre dans la table de hachage reste établi à environ 30% de l'espace total. Cette propriété est obtenue en choisissant le dividende de sorte que `n/d ≥ 0,7` et `d` premier.

1. Déterminer une valeur de d.

2. Écrire l'algorithme de parcours du fichier qui établit la correspondance entre la clé et le nom d'un client.

3. Écrire l'algorithme qui permet de rechercher l'article associé à un client en utilisant son nom.

© ENI Editions - All rigths reserved

Problèmes

Cette section présente des problèmes faisant intervenir des fichiers. On propose d'abord de formater un texte qui est la définition d'un algorithme. La section suivante pose un exercice de calcul sur des statistiques d'import/export. La troisième section montre comment exploiter les réponses à un questionnaire d'attitude et la section suivante en fait autant pour les réponses à une enquête d'utilité publique. Tous ces problèmes ne sont en fait que des prétextes pour écrire des algorithmes. Mais alors que précédemment, les algorithmes étaient indépendants, ici, ils doivent être composés pour obtenir une analyse complète du problème posé. De plus, les problèmes ne sont pas posés d'une façon académique. Ils nécessitent souvent d'être précisés ou complétés. Ainsi, aucun des exercices ne donne d'indication sur les structures à mettre en place et tous restent vagues sur la forme des données à traiter. En particulier, on ne précise jamais le type de fichier utilisé. Il faut donc faire un effort d'analyse et de réflexion pour obtenir une solution utilisable ensuite en programmation.

1. Formater un fichier source

On donne un fichier ne contenant que du texte. Ce fichier est un fichier qui définit un algorithme. Autrement dit, il s'agit d'un *fichier source* en « langage algorithmique ». Ainsi, chacun des enregistrements du fichier est une ligne d'instructions d'un algorithme. L'exercice demande de produire un nouveau fichier, imprimable, contenant les codes de mise en forme de certaines lignes de l'algorithme.

Exercice 7 : Mettre en forme un fichier source pour l'impression

Soit un fichier texte d'extension `.algo` par exemple. Il contient les instructions d'un algorithme quelconque, préparé dans une syntaxe quelconque à l'aide d'un éditeur de texte. L'éditeur produit un fichier ne contenant que des caractères standard du jeu de caractères de l'ordinateur utilisé. En particulier, il ne contient aucun enrichissement du texte.

On veut écrire un algorithme qui produit en sortie un fichier d'extension `.prn` mais qui contient, en plus du texte du fichier source, les codes destinés à l'imprimante utilisée et qui permettent d'enrichir le texte pour le présenter d'une façon plus agréable si ce n'est plus lisible. Ainsi, par exemple, les commentaires seront imprimés en caractères italiques, les mots clés du pseudo-langage utilisé seront imprimés en caractères bleus et en casse grasse…

Lorsque l'imprimante doit éditer des caractères particuliers, comme des caractères italiques par exemple, elle doit recevoir, avant les caractères concernés, une séquence de codes qui la bascule dans le mode d'impression correspondant. Une autre séquence de codes la replace dans le mode standard. Ainsi, pour telle petite imprimante de bureau, il faut envoyer la suite de codes <ESC>R<DC3> pour qu'elle imprime en caractères italiques et <ESC>R<VT> pour qu'elle reprenne la police des caractères normaux. <ESC> est un caractère de code particulier, par exemple 27. De même, <DC3> et <VT> sont respectivement les caractères de code 19 et 11, mais ce ne sont que des exemples.

1. Préparer la liste des enrichissements que vous envisagez pour le texte source des algorithmes.

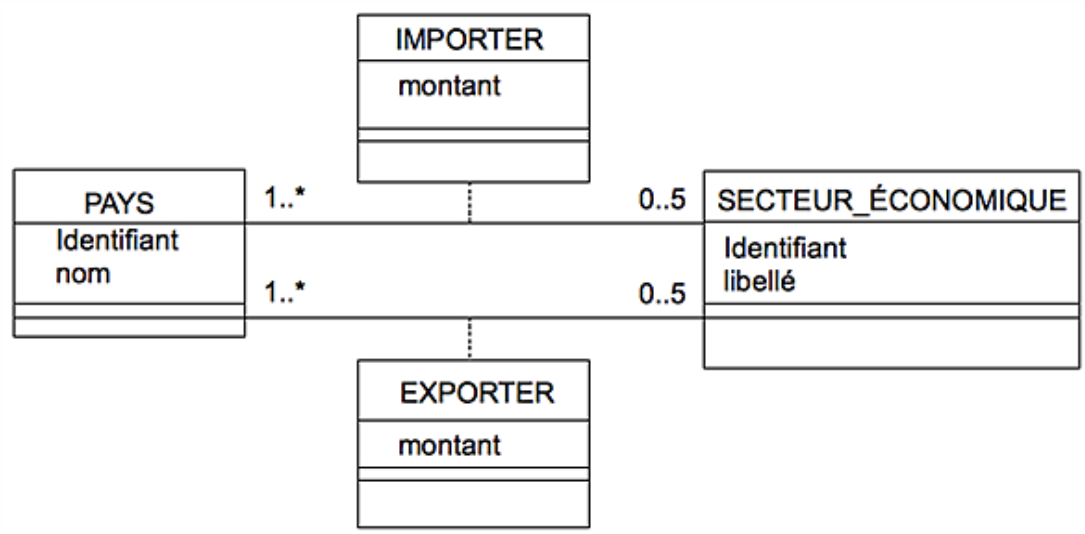
Il faut peut-être se limiter dans un premier temps à une seule transformation, par exemple celle des commentaires. On peut supposer, par exemple, qu'un commentaire est un texte signalé par l'une des suites de caractères '#', ou '//', ou '- -' lorsqu'il tient sur une seule ligne. Il est signalé par '(… *)' ou '/* ... */' lorsqu'il peut s'étendre sur plusieurs lignes. On peut se limiter au cas où un commentaire est seul sur sa ligne et où la suite des caractères qui l'identifie est toujours placée en début de ligne.*

2. Écrire l'algorithme de traitement du fichier source.

2. Statistiques d'import/export

On dispose d'informations relatives aux importations et exportations d'un ensemble de pays et on désire dépouiller ces informations pour en extraire un certain nombre d'indicateurs. Les informations qui concernent chaque pays sont les montants de ses importations et de ses exportations, selon 5 secteurs économiques au plus, exprimés en millions d'Euro. Les pays et les secteurs économiques sont identifiés par des nombres entiers. Le schéma conceptuel de la figure ci-dessous montre les associations entre les pays concernés et les secteurs économiques étudiés.

© ENI Editions - All rigths reserved



Ce schéma se lit de la façon suivante. Un pays importe dans 0 à 5 secteurs économiques étudiés. Pour chaque paire (pays ; secteur), l'association IMPORTER donne le montant des importations. Chaque secteur économique donne lieu à des importations pour 1 à plusieurs pays. La partie du schéma qui concerne les exportations se lit de la même façon.

Les informations recueillies ont été enregistrées dans quatre fichiers. Le premier est une table des pays. Le second est une table des secteurs économiques. Le troisième et le quatrième sont, respectivement, le fichier des importations et celui des exportations contenant, pour chacun et dans chaque enregistrement, les identifiants du pays et du secteur, suivi du montant correspondant.

Exercice 8 : Exploitation de statistiques économiques

1. Écrire un algorithme qui crée un fichier dont chaque enregistrement donne, pour un pays, les importations et les exportations par secteur.

Chaque enregistrement comporte, en plus, le total et la moyenne arithmétique pour les importations et les exportations. Un enregistrement supplémentaire donne les totaux et les moyennes pour chaque secteur économique, tous pays confondus.

2. Écrire un algorithme qui crée un fichier exprimant les bilans pour chaque pays par secteur. Le bilan est la différence des montants entre exportations et importations, y compris les totaux.

3. Écrire l'algorithme qui prend en entrée le nom d'un pays et qui retourne les identifiants des secteurs économiques où le montant des importations et des exportations sont les plus élevés pour ce pays.

4. Déterminer pour chaque secteur le pays qui réalise les meilleures performances à l'exportation.

3. Exploiter un questionnaire d'attitude

On a soumis des sujets à un questionnaire d'attitude. Ce questionnaire comporte six items. Les sujets devaient répondre à une question par item et nécessairement par OUI ou par NON. Chaque item était formulé de sorte qu'une réponse positive exprime toujours un accord avec l'attitude mesurée. Les réponses ont été rassemblées dans un fichier dont chaque enregistrement contient un identifiant numérique du sujet interrogé et les réponses de ce sujet à chacune des questions posées.

Exercice 9 : Questionnaire d'attitude

1. Écrire un algorithme qui constitue un tableau des fréquences croisées des réponses aux items pris deux à deux.

2. Écrire un algorithme qui reclasse les données en fonction du nombre de réponses positives par sujet et par item.

Pour cela, on calculera le score par individu en effectuant la somme des résultats par enregistrement. Les enregistrements seront ensuite reclassés par score décroissant. Le score par item sera alors calculé et ces scores seront reclassés par score croissant.

4. Exploiter les réponses à une enquête d'utilité publique

Dans le cadre d'une enquête d'utilité publique, des sujets ont été soumis à un questionnaire comportant huit questions relatives à l'implantation d'une centrale nucléaire. Les réponses positives, c'est-à-dire favorables, ont été codées 1 et les réponses défavorables ont été codées 0. Chaque personne interrogée est repérée par un identifiant

© ENI Editions - All rigths reserved

numérique.

Exercice 10 : Enquête d'utilité publique

On désire réaliser une pré-exploitation des résultats sur les cinquante premiers enregistrements.

1. Écrire l'algorithme qui constitue en mémoire un tableau des réponses aux questions d'attitude.

Dans ce tableau, chaque ligne correspond à une personne interrogée et chaque colonne à une réponse à une question d'attitude.

2. Dénombrer les réponses favorables à chaque question.

3. Écrire l'algorithme qui affiche à l'écran l'histogramme des fréquences associées aux résultats de la question précédente.

4. Calculer, pour chaque personne, une note globale d'attitude égale au nombre de réponses favorables qu'elle a données.

5. Dénombrer les personnes ayant obtenu une note globale donnée puis afficher l'histogramme des fréquences associées à chacune des notes globales obtenues.

On définit la distance entre deux personnes comme le nombre de réponses différentes qu'elles ont données aux questions posées.

6. Écrire l'algorithme qui calcule la distance entre deux personnes quelconques. Déterminer la distance minimum relevée et la liste des couples dont la distance est minimum.

© ENI Editions - All rigths reserved

Notes bibliographiques

Comme dans d'autres domaines, [KNUTH73] est encore une référence en ce qui concerne les fichiers. Les livres [CB84], [SED91] et [MB84] sont un peu anciens mais ils restent des livres intéressants pour l'étude des méthodes algorithmiques et de programmation du domaine.

Les exercices de la section Problèmes sont inspirés d'une source dont je ne retrouve pas les références. *Dixi et salvavi animam meam.*

© ENI Editions - All rigths reserved

Résumé

Les fichiers permettent d'assurer la persistance des données. Pour cela, elles sont enregistrées sur des supports externes d'où elles peuvent être récupérées. Ce chapitre a présenté l'organisation et l'accès séquentiel puis quelques éléments concernant l'organisation et l'accès sélectif. Quelques problèmes permettent de s'exercer à l'algorithmique qui utilise ces fichiers.

© ENI Editions - All rigths reserved

Bibliographie

[CB84] G. CLAVEL, J. BIONDI : *Introduction à la programmation - Tome 2 : Structures de données* ; MASSON, 1984.

[KNUTH73] Donald KNUTH : *The Art of Computer Programming, Vol 3 - Sorting and Searching* ; ADDISON-WESLEY, 1973.

[MB84] Bertrand MEYER, Claude BAUDOIN : *Méthodes de programmation* ; EYROLLES, 1984.

[SED91] Robert SEDGEWICK : *Algorithmes en langage C* ; INTERÉDITIONS, 1991.

© ENI Editions - All rigths reserved

Introduction

Ce chapitre s'intéresse à des techniques qui permettent de faire des *prévisions*. Bien que les notions abordées restent élémentaires, le contenu est essentiellement mathématique. Les lecteurs qui n'ont pas le goût de pratiquer ces activités peuvent passer au chapitre suivant.

Il existe des prévisions de nature et de portée différentes. On peut s'intéresser à des événements qui se produiront inévitablement dans le futur. C'est le domaine, par exemple, des prévisions météorologiques ou des méthodes de calcul qui cherchent à déterminer le parcours d'un astre sur sa trajectoire. On dispose d'un modèle mathématique qui décrit, plus ou moins bien, la réalité des phénomènes et qui permet de connaître à l'avance, dans une certaine mesure, leur évolution. On peut alors prendre des décisions qui régleront notre comportement. On peut vouloir faire des prévisions sur l'évolution d'un phénomène qu'il est impossible de mettre réellement en œuvre. Comment se comportera la faune de tel cours d'eau en cas de pollution massive par des hydrocarbures ? Quelles sont les conséquences d'un accident d'avion sur tel quartier de telle ville ? Les phénomènes dont il s'agit sont *destructifs* et ils ne peuvent être provoqués volontairement. L'évolution au cours du temps d'un processus dynamique, mais qui se déroule sur une échelle de longue durée par rapport à la durée de vie humaine, fait aussi l'objet de méthodes de prévisions. Elles consistent à « accélérer » le déroulement du temps. C'est le cas, par exemple, quand on veut prévoir comment évoluera un écosystème, quelles seront les conséquences d'une activité humaine sur un biotope ? La réalisation de calculs prédictifs sur des phénomènes statistiques ou chaotiques ou, tout simplement, la réalisation des estimations statistiques de résultats déterministes mais trop difficiles à calculer font aussi partie du champ des préoccupations de ces prévisions. Finalement, nous pouvons, en première approximation et surtout pour simplifier, classer les phénomènes dont on veut prévoir les évolutions en deux catégories :

- les *processus déterministes* dynamiques dont l'évolution est décrite par des lois physiques connues. La programmation de ces lois permet d'en étudier l'évolution :
    - prévisions météorologiques ;
    - trajectoire d'un astre ;
    - calcul d'intégrales en physique atomique ;
    - etc.
- les *processus aléatoires* qui évoluent d'une façon statistique. Il s'agit alors de pouvoir « imiter le hasard » pour pouvoir prédire.

Ce chapitre présente un ensemble de problèmes qui illustrent ces deux catégories de simulations. On met d'abord en place, en section Générer des nombres pseudo-aléatoires, les « outils logiciels » qui permettront ensuite de rentrer dans le vif du sujet. La section Jeux de hasard aborde la simulation de quelques jeux de hasard simples. La simulation de processus dynamiques est envisagée en section Simulation de processus dynamiques. Nous étudions ainsi des exemples simples de courses poursuites, de propagation d'une rumeur dans une population, etc. La simulation de processus aléatoires regroupe deux types de problèmes. Nous montrons d'abord en section Simulation statistique de phénomènes déterministes comment obtenir une estimation statistique du résultat de problèmes déterministes, comme le calcul de π, le calcul d'intégrales définies, etc. Nous montrons ensuite en section Simulation de phénomènes aléatoires comment simuler des processus totalement aléatoires, comme la propagation d'une rumeur dans une population ou l'influence de la dispersion des caractéristiques des composants d'un circuit électronique sur le fonctionnement de ce circuit.

© ENI Editions - All rigths reserved

Générer des nombres pseudo-aléatoires

Dans cette section, nous étudions la génération de nombres pseudo-aléatoires qui seront ensuite utilisés dans les autres sections. Nous voulons disposer d'une suite de nombres obtenue « au hasard ». Pour l'instant, cette expression signifie que la connaissance de tous les exemplaires des nombres déjà obtenus ne permet pas de déterminer la valeur du prochain nombre à obtenir. Ainsi, par exemple, quand on dispose d'une pièce de monnaie bien équilibrée et que nous l'utilisons pour jouer à « pile ou face », la suite des issues déjà obtenues ne permet pas de connaître le résultat du prochain jet. Tout ce que nous pouvons dire est que ce résultat sera PILE avec la probabilité 0,5 et FACE avec la même probabilité. La suite des issues obtenues est aléatoire.

Préparer un générateur automatique de nombres aléatoires est impossible. Cependant, il existe des techniques de calcul qui permettent d'obtenir des suites qui « imitent » des nombres obtenus « au hasard », au sens défini précédemment. Comme ces générateurs engendrent les nombres à l'aide d'une fonction déterministe, ils sont qualifiés de « pseudo-aléatoires ». La première section étudie quelques exemples simples de générateurs. Cependant, obtenir une suite automatique de nombres ne suffit pas. Il faut disposer de tests qui permettent de s'assurer que les suites obtenues « imitent bien le hasard », autrement dit qu'elles ont les qualités requises pour se substituer aux suites de nombres parfaitement aléatoires. La deuxième section montre comment réaliser quelques tests simples sur de telles suites.

1. Quelques générateurs

Pour générer des nombres « au hasard », on engendre une suite de nombres $(x_n)_{n>0}$ à partir d'un nombre initial donné x_0. Une fonction mathématique f permet de calculer $x_n = f(x_{n-1})$. On obtient donc une suite de nombres pseudo-aléatoires puisque la connaissance d'un élément x_i quelconque de cette suite permet de déterminer tout élément de la suite. Quelques tests de la prochaine section montreront pourquoi et quand on peut accepter que de telles suites se substituent à des nombres au hasard. Suivant la fonction f et la racine x_0 de la suite, on obtiendra des résultats évidemment différents, mais de plus ou moins « bonnes qualités ». Il existe une grande variété de choix possibles et générer des nombres pseudo-aléatoires de qualité est encore un domaine de recherche. Cette section propose quelques exemples simples.

Dans tous les cas, le générateur utilise *une graine*. C'est le nombre x_0 qui commence la suite. Ce nombre conditionne la valeur des éléments suivants et donc, en partie, les qualités du générateur. Il doit être choisi « au hasard », ce qui est impossible. Cependant, là encore, nous pouvons utiliser des artefacts qui permettent de simuler un tel choix. Nous pouvons, par exemple, utiliser une source naturelle de nombres aléatoires, comme une source radioactive ou une suite de jets d'une pièce de monnaie par exemple. On sait pourtant que la plupart des sources naturelles fournissent des données biaisées ou corrélées. Cela signifie soit que la suite obtenue est périodique, soit que certaines issues sont plus fréquentes que d'autres. Dans la suite, nous supposons qu'une fonction réelle **racine** sans paramètre nous fournit en résultat un nombre aléatoire qui nous servira de graine pour le générateur. Sa spécification est celle de l'algorithme ci-dessous.

*Algorithme 1 : Spécification de la fonction **racine***

```
Algorithme racine
    # Un réel aléatoire de ]0, 1[.
Résultat : RÉEL
postcondition
    0 < Résultat < 1
    Résultat est un nombre aléatoire.
fin racine
```

L'appel de cette fonction permet donc d'obtenir une graine et le générateur permet d'engendrer la suite des nombres pseudo-aléatoires attendus. L'utilisation de la même graine avec le même générateur redonne systématiquement la même suite, ce qui permet, en programmation, de procéder aux tests des applications qui utilisent les nombres de la suite.

Cette section étudie des générateurs qui engendrent une suite de nombres *uniformément répartis* sur l'intervalle `]0,1[`. Ici, nous nous contenterons de l'idée intuitive qui explique cette expression : chacun des réels de l'intervalle « a la même probabilité », que chacun des autres nombres de l'intervalle, d'être obtenu par le générateur.

a. Le 147-générateur

Le *147-générateur* est une idée de [RAD77]. Étant donné un réel x_{i-1} quelconque de la suite, l'élément suivant x_i est obtenu en multipliant x_{i-1} par le nombre 147 et en extrayant la partie décimale du produit. Le nombre 147 est choisi empiriquement à partir des qualités des suites obtenues pour différentes valeurs. La suite est donc définie par :

© ENI Editions - All rigths reserved

$$\begin{cases} x_0 = racine() \\ x_i = fraction(147 \times x_{i-1}), i > 0 \end{cases}$$

La fonction **fraction** rend la partie décimale (fractionnaire) du réel qu'elle reçoit en paramètre. La fonction **racine** permet d'obtenir « au hasard » la graine du générateur.

Voici quelques exemples de nombres obtenus avec ce générateur :

x_0 = 0, 141 592 654 => 147 . x_0 = 20, 814 120 14
x_1 = 0, 814 120 14 => 147 . x_1 = 119, 675 660 6
x_2 = 0, 675 660 6 ...
...

La spécification de la fonction fraction est la suivante :

*Algorithme 2 : Spécification de **fraction***

```
Algorithme fraction
    # La partie décimale de x.
Entrée
    x : RÉEL
Résultat : RÉEL
précondition
    x ≥ 0
postconditionRésultat = x - [x]
fin fraction
```

Étant donné un élément x quelconque de la suite engendrée à partir de la graine, on obtient l'élément suivant à l'aide de la fonction **hasard** définie par :

Algorithme 3 : Fonction du 147-générateur

```
Algorithme hasard# Élément qui suit x dans la suite aléatoire.
Entrée
    x : RÉEL
Résultat : RÉEL
précondition
    Résultat <- fraction(147 x x)
postcondition
    Résultat = fraction (147 x x)
fin hasard
```

Cette fonction est ensuite appelée itérativement pour obtenir les éléments successifs de la suite. Voici, à titre d'exemple, comment engendrer 100 nombres réels pseudo-aléatoires uniformément répartis sur]0,1[:

```
...
constante
    MAX : ENTIER <- 100 # Longueur de la suite à produire.
variable
    graine : RÉEL   # La graine du générateur.
    aléa   : RÉEL# Le nombre aléatoire obtenu.
    i      : ENTIER # Numéro dernier nombre obtenu dans la suite.
initialisation
    graine <- racine()       # Initialise la suite.
    aléa   <- hasard(graine) # Génère le premier nombre.
    i <- 1
jusqu'à
    i > MAX
répéter
    faire quelque chose avec le nombre obtenu.
    aléa <- hasard(aléa)     # Nombre aléatoire suivant.
    i <- i + 1
```


© ENI Editions - All rigths reserved

```
fin répéter
...
```

Les sections qui traitent des simulations montreront comment transformer et utiliser les suites produites pour en faire « quelque chose d'utile ».

Le générateur qui vient d'être présenté a surtout le mérite d'être simple à implémenter et à utiliser. Il ne faut pourtant pas en attendre des miracles. Les qualités des suites obtenues sont plutôt moyennes et elles sont sensibles à la graine. En particulier, la suite est toujours périodique, parfois avec une période si faible qu'elle en devient inutilisable. Le générateur suivant est bien meilleur, sans être parfait.

b. Générateurs de Hamming

Ces générateurs forment une famille. Ils obtiennent tous leurs suites à partir d'une formule de récurrence de la forme :

$$\begin{cases} e_0 = racineEntière() \\ e_i = a \times e_{i-1} + b \; modulo \; m \end{cases}$$

dans laquelle les nombres utilisés sont tous des entiers et *modulo* est l'opérateur qui rend le reste de la division euclidienne de son premier argument par le second. Les réels uniformément répartis sur]0,1[sont alors obtenus par la division réelle de e_i par un entier c bien choisi. La fonction **racineEntière** joue le même rôle que la fonction racine mais en retournant un entier aléatoire dans le domaine des entiers représentables par la machine utilisée.

C'est le choix des valeurs e_0, a, b, c et m qui conditionne les qualités propres de la suite. On peut montrer qu'un bon générateur est obtenu pour $m = 2^{n-1}$ dans lequel n est le nombre de bits par mot du calculateur utilisé. Ainsi, m n'est autre que le plus grand entier représentable en machine. On choisit alors a de la forme $a = 8 \times k \pm 3$ où k est un entier quelconque, $b = 0$ et $c = m - 1$. De plus, le nombre e_0 initial doit être un entier positif impair, inférieur à m. On peut alors démontrer que e_i est obtenu par :

```
...
e_i <- a x x_i - 1
si
    e_i < 0
alors
    e_i <- e_i - m
fin si
...
```

Les calculs ci-dessus supposent que l'arithmétique se fait modulo m en machine. La procédure **hasard** déduite de ces formules est donnée par l'algorithme ci-dessous. Elle utilise une structure de données définie par :

```
type
    HASARD
structure
    e : ENTIER # Racine de Hamming.
    x : RÉEL   # Aléa généré.
invariant
    0 < x < 1
fin hasard
```

La variable e contient la dernière valeur obtenue dans la suite des entiers de Hamming. Le nombre x est le nombre aléatoire attendu.

Algorithme 4 : Définition de la procédure du générateur de Hamming

```
Algorithme hasard
    # Élément suivant x dans la suite aléatoire de Hamming de base
    # aléa.e.
Entrée
    aléa : HASARD
constante
```

© ENI Editions - All rigths reserved

```
    k : ENTIER <- valeur choisie pour k
    a : ENTIER <- 8 x k + 3
    n : ENTIER <-  nombre de bits du mot machine
    m : ENTIER <- 2^n - 1 # Plus grand entier représentable en machine.
réalisation
    aléa.e <- a x aléa.e
    si
        aléa.e < 0
    alors
        aléa.e <- aléa.e - m
    fin si
        aléa.x <- aléa.e / (m - 1)
postcondition
    aléa.e est la nouvelle valeur de la racine de Hamming
    aléa.x est le nombre aléatoire généré
fin hasard
```

Cependant, cette méthode utilise une arithmétique modulo m. Autrement dit, on suppose que lorsqu'un nombre atteint la valeur du plus grand entier représentable en machine, lui ajouter 1 ne provoque pas d'erreur, ce qui n'est pas vrai quand on utilise un langage de programmation moderne. Préparer un générateur utilisable dans ce contexte nous conduirait trop loin. Les connaissances requises dépassent de toute façon les objectifs et les ambitions de ce livre. Pour ceux qui voudraient expérimenter et programmer effectivement les algorithmes proposés, on peut leur conseiller de lire le manuel du langage de programmation adopté. Tous les langages modernes proposent une fonction qui permet de générer des nombres aléatoires.

Revenons à nos générateurs.

Il existe ainsi de nombreuses façons de générer des nombres pseudo-aléatoires. Dans la suite, nous utilisons un type de données **ALÉA** défini par :

```
type
    ALÉA
structure
    ...
    x : RÉEL
    ...
invariant
    0 < x < 1
fin ALÉA
```

Les opérations applicables sur une donnée a de type **ALÉA** sont :

- **initialiser**(a : **ALÉA**) est une procédure qui génère la graine et initialise le générateur ;
- **suivant**(a : **ALÉA**) est une procédure qui génère le nombre suivant dans la suite engendrée par le générateur ;
- **item**(a : **ALÉA**) est une fonction qui rend le réel aléatoire généré de]0,1[.

Des appels répétés à **item** sans appeler d'abord **suivant** retournent systématiquement le même nombre. Pour générer 100 réels aléatoires, on écrit alors :

```
...
constante
    MAX : ENTIER <- 100 # Longueur de la suite.
variable
    hasard : ALÉA
    nombre : RÉEL       # Le nombre aléatoire obtenu.
    i      : ENTIER     # Nombre de réels déjà engendrés.
initialisation
    initialiser(hasard) # Initialise la suite.
    i <- 1              # Un nombre généré.
jusqu'à
    i > MAX
répéter
    suivant(hasard)     # Élément suivant de la suite.
    nombre <- item(hasard)
```


© ENI Editions - All rigths reserved

```
    utiliser nombre pour en faire quelque chose
    i <- i + 1
fin répéter
...
```

Pour expérimenter en programmant, on peut utiliser de cette façon l'un des générateurs définis précédemment ou celui qui est implémenté dans tout calculateur moderne. Ainsi, par exemple, la fonction **urandom** d'une machine qui exécute UNIX permet d'obtenir des suites de nombres aléatoires de qualité satisfaisante.

Exercice 1 : Primitives de génération de nombres aléatoires

1. Écrire les primitives de génération de nombres aléatoires utilisant le 147-générateur.

2.Même question avec le générateur de Hamming.

2. Tester une suite de nombres pseudo-aléatoires

On donne un générateur de nombres aléatoires. *Ce générateur simule-t-il le hasard d'une façon acceptable ?* Considérons un générateur qui a permis d'obtenir une suite de chiffres binaires, 0 et 1, pour simuler le lancer d'une pièce bien équilibrée et sans biais. On s'attend à ce que la suite contienne autant de 0 que de 1, mais cela ne suffit pas. Ainsi, par exemple, si ce générateur donne, pour dix lancers, la suite 0000011111, nous ne pouvons pas prétendre que cette suite est aléatoire. Nous devons donc disposer de tests qui vérifient que la suite obtenue possède les mêmes propriétés qu'une suite de nombres obtenue « au hasard ». Voici quelques tests pour un générateur qui engendre des nombres uniformément répartis sur]0,1[:

- test de la moyenne : la moyenne des nombres obtenus doit être égale à 0,5 ;
- test de la variance : la variance doit être égale à 1/12 ≈ 0, 083 333 ;
- test de l'histogramme : l'histogramme doit être plat ;
- test du χ^2 (Khi-deux).

Le test de l'histogramme consiste à déterminer, pour chaque chiffre de 0 à 9, la fréquence absolue et la fréquence relative. La fréquence relative doit évidemment être égale à 0,1. Pour obtenir les chiffres, il suffit, par exemple, de multiplier chaque nombre aléatoire de]0,1[par 10 et d'extraire la partie entière du produit :

```
...
suivant(hasard)                 # Générer l'élément suivant.
nombre  <- item(hasard)         # Récupérer le nombre aléatoire.
chiffre <- entier(10 x nombre) # Calculer le chiffre correspondant.
...
```

La variable chiffre contient ainsi un entier entre 0 et 9. Il peut servir à adresser la case correspondante d'un tableau qui contient les fréquences absolues des différents chiffres.

On peut aussi engendrer les nombres dans]0,1[et ne retenir, pour chaque nombre, que certains chiffres de la partie décimale. Ainsi, par exemple, on peut retenir les cinq premiers chiffres de la partie décimale de chaque nombre obtenu à l'aide du 147-générateur. Reprenons notre exemple :

x_0 = 0, 141 592 654 => 147 x x_0 = 20, 814 120 14
x_1 = 0, 814 120 14 => 147 x x_1 = 119, 675 660 6
x_2 = 0, 675 660 6 ...
...

On ne retiendra alors que les chiffres 81412 67566… comme suite de chiffres aléatoires.

Soit n le nombre d'épreuves, c'est-à-dire le nombre d'entiers aléatoires obtenus. Si on s'intéresse aux chiffres en base dix, par exemple, la suite précédente a permis d'obtenir n = 10 chiffres en deux blocs de cinq chiffres. Chacun des nombres obtenus prend une valeur v entre 0 et m - 1 : $0 \leq v < m$. Pour notre exemple, m = 10 et donc, chaque chiffre prend une valeur entre 0 et 9.

On dispose ainsi, pour chaque tirage aléatoire, de m issues possibles de valeurs 0, 1, 2, …, m – 1. Chaque issue i parmi les m issues possibles est apparue un nombre de fois égal à f_i appelée la *fréquence absolue de l'issue i*. Ainsi, par exemple, sur n = 1000 tirages aléatoires de nombres dans [0,10[, on aura obtenu peut-être 97 fois le chiffre 6 et alors

© ENI Editions - All rigths reserved

$f_6 = 97$.

Le test du Khi-deux consiste à vérifier que la distribution des résultats est « réaliste », c'est-à-dire qu'elle imite bien une distribution de résultats aléatoires. Ainsi, dans l'exemple précédent, nous devrions obtenir une fréquence f_i égale à 100 pour chacune des issues i ; mais si nous voulons que la suite ait les propriétés d'une suite aléatoire, nous ne devons pas obtenir des fréquences toutes exactement égales à la valeur théorique 100. L'obtention d'un tel résultat parfait devrait faire suspecter un générateur biaisé. Pour réaliser le test, on calcule :

$$\chi^2 = \frac{\sum_{i=0}^{i=m-1} \left(f_i - \frac{n}{m}\right)^2}{\frac{n}{m}}$$

La valeur à obtenir doit être « proche » de `m`. Il faut encore donner un sens à l'expression « proche » de `m`. En pratique, on admet que la suite est acceptable lorsque χ^2 est à une distance de `m` inférieure au double de la racine carrée de `m`, soit :

$$m - 2\times\sqrt{m} < \chi^2 < m + 2\times\sqrt{m}$$

Ainsi, toujours pour le même exemple, nous devons obtenir :

$$10 - 2\times\sqrt{10} < \chi^2 < 10 + 2\times\sqrt{10}$$

soit $3{,}68 < \chi^2 < 16{,}32$.

Exercice résolu 1 : Tests d'un générateur de nombres pseudo-aléatoires

Écrire les algorithmes permettant de réaliser les premiers tests d'un générateur.

Solution

Cette solution ne présente que l'algorithme permettant d'obtenir le nombre χ^2 car c'est le seul calcul qui soulève quelques difficultés.

On peut montrer facilement que la formule de calcul se met sous la forme :

$$\chi^2 = \frac{m}{n} \times \sum_{i=0}^{i=m-1} \left(f_i^2 - n\right)$$

Le calcul est réalisé par l'algorithme suivant :

Algorithme 5 : Calcul du χ^2

```
Algorithme khi2
    # Le khi2 des composantes de f.
Entrée
    f : TABLEAU[RÉEL] # Tableau des fréquences absolues.
    n : ENTIER        # Nombre de valeurs (issues) des tirages.
    m : ENTIER        # Nombre d'issues.
Résultat : RÉEL
...
variable
    somme : ENTIER # La somme des carrés des fréquences.
    i     : ENTIER # Le numéro de la prochaine case à additionner.
    produit : RÉEL# khi2 x n.
    rn      : RÉEL# n converti en réel.
initialisation
```


© ENI Editions - All rigths reserved

```
    somme <- 0
    i     <- 0
jusqu'à
    i ≥ m
    invariant
        i = 0 => somme = 0
        i > 0 => somme = (f[0]² + f[1]² + … f[i - 1]²)
    variant de contrôle
        m - i
répéter
    somme <- somme + f[i] x f[i]
    i <- i + 1
fin répéter
produit <- réel(somme x n)
rn      <- réel(n)
Résultat <- (produit - 1) / rn
...
fin khi2
```

La fonction **réel** est une fonction de conversion de type. Elle transforme un entier en nombre réel de sorte que la division soit la division des réels et non la division euclidienne.

Il existe de nombreux tests de générateurs et ceux qui précèdent sont parmi les plus simples. Il n'est évidemment pas question d'étudier ce domaine ici, mais un dernier test, le *test du poker*, permettra de déterminer d'autres caractéristiques du générateur choisi.

Le test consiste à regrouper les chiffres générés par blocs de 5. La table ci-dessous donne les probabilités théoriques des différentes figures obtenues lorsque le générateur est parfait.

Figure	***Probabilité***
5 chiffres différents	0,302 4
1 paire	0,504 0
2 paires	0,108 0
brelan	0,072 0
full	0,009 0
carré	0,004 5
5 chiffres identiques	0,000 1

Le *brelan* est constitué de trois chiffres identiques. Le *full* est constitué d'un *brelan* et d'une paire. En base dix, il existe 10^5 tranches de cinq chiffres et chacune a la probabilité 10^{-5} d'apparaître.

Exercice 2 : Test du poker

Écrire les algorithmes permettant de remplir un tableau donnant, pour chaque figure du test du poker appliqué à un générateur, la probabilité théorique de la figure, la fréquence absolue et la fréquence relative de son apparition.

La solution de l'exercice précédent n'est pas simple, notamment parce que les motifs à rechercher dans chaque tranche de cinq chiffres ne sont pas fixés. On peut passer directement à l'exercice suivant pour une initiation.

Exercice 3 : Un autre générateur

On obtient des nombres pseudo-aléatoires en engendrant les nombres de la suite définie par :

$$\begin{cases} 0 < x_0 < 1 \\ x_n = fraction\left((\pi + x_{n-1})^5\right), n > 0 \end{cases}$$

© ENI Editions - All rigths reserved

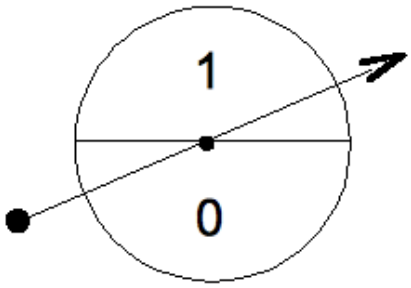
1. *Écrire les algorithmes qui implémentent ce générateur.*
2. *Écrire les algorithmes de tests.*

© ENI Editions - All rigths reserved

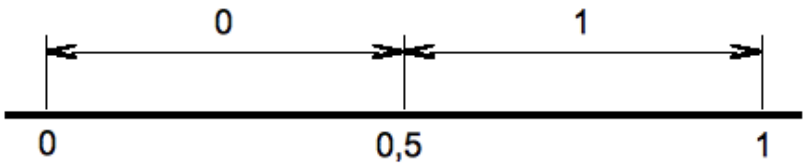
Jeux de hasard

1. Simuler une roulette

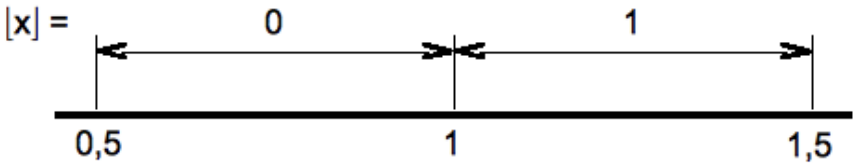
Considérons la roulette schématisée par le dessin de la figure ci-dessous.



Elle symbolise le fait que les deux issues {0,1} ont la même probabilité 0,5 de sortir. On sait obtenir des nombres aléatoires uniformément répartis sur l'intervalle]0,1[depuis la section précédente. *Comment obtenir des nombres entiers aléatoires dans {0,1}, chaque issue ayant la même probabilité que toute autre ?* Les réels x que nous savons obtenir sont tels que 0 < x < 1. Il suffit donc de sortir 0 pour toute valeur de x entre 0 et 0,5 ou 1 pour toute valeur de x entre 0,5 et 1. La figure ci-dessous schématise ce que nous voulons obtenir.



Les nombres générés sont équiprobables. Par conséquent, x appartient à l'intervalle]0 ; 0,5] avec la probabilité 0,5 et à l'intervalle [0,5 ; 1[avec la même probabilité. Si on ajoute 0,5 à x, le résultat appartient à l'intervalle]0,5 ; 1,5[et la partie entière du résultat vaut 0 ou 1 avec la probabilité 0,5. C'est ce que nous voulons. La figure suivante représente cette situation.

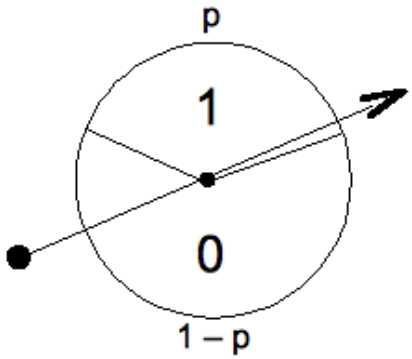


Soit **entier** la fonction qui retourne la partie entière de son argument. Le principe algorithmique de simulation d'une roulette équitable est donc simple :

```
...
variable
    a     : ALÉA# La structure maintenue par le générateur.
    x     : RÉEL   # Un exemplaire du réel aléatoire généré.
    issue : ENTIER # Une valeur entière générée.
...
# Initialiser le générateur.
initialiser(a)
# Générer un réel aléatoire dans ]0, 1[.
suivant(a)
# obtenir un exemplaire du réel généré.
x <- item(a)
# Faire tourner la roulette équitable.
issue <- entier(x + 0,5)
...
```

Une roulette n'est pas équitable lorsque les issues n'ont pas la même probabilité. Considérons la figure suivante :

© ENI Editions - All rigths reserved



Elle donne l'issue 1 avec la probabilité p et l'issue 0 avec la probabilité 1 - p. Pour obtenir l'issue 1 avec la probabilité p, il suffit de calculer la partie entière de x + p, avec 0 < p < 1.

Exercice 4 : Estimation d'une probabilité par simulation

On considère les deux roulettes précédentes.

1. Écrire les algorithmes des tests des roulettes.

Ceux qui programment effectivement pourront essayer p=0,4 puis p=0,1 pour la roulette de la figure ci-dessus.

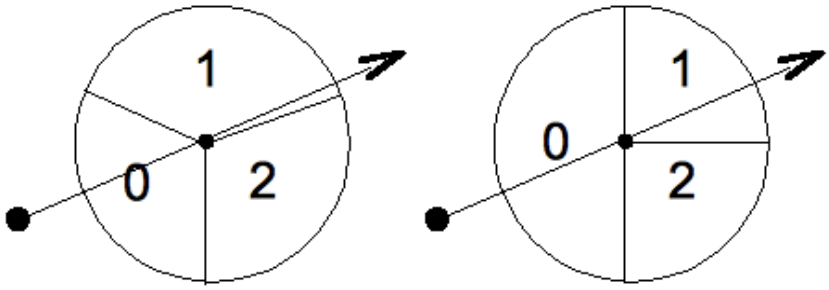
On fait tourner ces deux roulettes autant de fois qu'il est nécessaire pour estimer la probabilité que la première donne un 1 avant la seconde, c'est-à-dire que la première donne 1 alors que l'autre donne 0. Pour cela, on fait tourner chaque roulette, à tour de rôle, 1000 fois et on « note » les résultats : l'estimation de la probabilité cherchée et le rapport nombre d'issues favorables divisé par le nombre d'essais (ici 1000). Ainsi, par exemple, si j'ai obtenu que la première donne 1 alors que l'autre donne 0 dans 32 cas, la probabilité cherchée est 0,032.

2. Écrire l'algorithme de cette simulation.

Pour obtenir des nombres réels aléatoires uniformément répartis sur un intervalle]a, b[, on calcule (b - a) x x + a. On parle alors de distribution rectangulaire.

Exercice 5 : Simulation d'une roulette à trois issues.

On veut simuler les roulettes des figures ci-dessous.



[roulette de gauche] [roulette de droite]

1. Écrire pour chacune d'elles l'algorithme qui permet de simuler un tirage.

2. Écrire l'algorithme d'une simulation de chacune d'elles pour obtenir les fréquences des différentes issues.

Remarquez que, pour la roulette de gauche, il s'agit d'engendrer des nombres dans l'intervalle]0, 3[. Il suffit donc de calculer la partie entière de 3 x x puisque, ici, b = 3 et a = 0. Pour la roulette de droite, on calcule la somme des parties entières des nombres x + 0,5 et x + 0,25.

2. Simuler un dé

Exercice 6 : Jeu de dés à deux joueurs

Il s'agit de jouer contre une machine. M est la machine et elle joue contre N. M lance un dé bien équilibré. C'est alors au tour

© ENI Editions - All rigths reserved

de N de jouer. Mais avant de lancer son dé, N doit prédire s'il fera mieux ou moins bien que M. Si N fait une bonne prédiction, il marque un point ; sinon, c'est M qui marque le point.

1. Existe-t-il une stratégie optimale pour le joueur N ?

N joue selon la stratégie optimale reconnue à la question précédente.

2. Estimer par simulation la probabilité qu'a N de gagner.

L'algorithme doit donner le nombre de points gagnés par chaque joueur à chaque tour.

3. (Question hors contexte pour ce livre) Faire jouer deux machines l'une contre l'autre.

© ENI Editions - All rigths reserved

Simulation de processus dynamiques

Cette section étudie deux exemples dont la nature devrait permettre de comprendre en quoi les méthodes des sections suivantes sont originales. Ces deux exemples n'apparaissent donc que comme « faire valoir » des sections suivantes. Le premier exemple étudie la propagation d'une rumeur dans une population. Le second montre comment représenter un cas simple de course poursuite.

1. Propagation d'une rumeur

Soit une population humaine homogène de n individus dans laquelle une rumeur se propage de bouche à oreille. On suppose que toute personne ayant connaissance de la rumeur la propage jusqu'à ce qu'elle rencontre une personne qui la connaît déjà. Elle cesse alors de la propager. Nous nous intéressons à l'évolution des effectifs des différentes classes d'individus dans la population. *La rumeur s'éteindra, mais atteint-elle tout le monde ? Quel pourcentage de la population reste étranger à cette rumeur ?*

Pour réaliser une simulation dynamique de la propagation, on fait des hypothèses sur la fréquence des rencontres de chaque type d'individu. Posons `ignore(t)` l'effectif de la population qui ignore la rumeur à l'instant `t`. On note, de même, `connaît(t)` l'effectif de ceux qui la connaissent mais ne la répandent plus et `répand(t)` l'effectif de ceux qui la répandent. À tout instant `t`, on a :

```
connaît(t) = n - (ignore(t) + répand(t))
```

Pendant un intervalle de temps (une durée) Δt, les effectifs `ignore(t)` et `répand(t)` évoluent en `ignore(t+Δt)` et `répand(t+Δt)`. Les contagions se produisent lors de rencontres de type ignore <-> répand qui sont au nombre de `ignore(t) x répand(t)` et la variation des effectifs correspondants pour un petit Δt est proportionnelle à ce produit. Nous pouvons donc écrire :

```
ignore(t+Δt) = ignore(t) - a x ignore(t) x répand(t)
```

où `a` est un coefficient de proportionnalité constant, fixé d'avance.

Les immunisations se produisent lors de rencontres de type (répand <-> répand) ou (répand <-> connaît). Les nombres de telles rencontres sont, respectivement, de `répand(t) x 0,5 x (répand(t) - 1)` et `répand(t) x connaît (t)`. Dans une rencontre de type (répand <-> répand), deux personnes passent simultanément dans le camp `connaît`. La variation du nombre d'immunisations est donc proportionnelle à :

```
I = répand(t) x connaît(t) + 2 x répand(t) x 0,5 x (répand(t) - 1)
```

On peut facilement démontrer que `I` est donné par :

```
I = répand(t) x (n - ignore(t) - 1)
```

Pendant une durée Δt, le flux provenant de `ignore` fait augmenter `répand(t)` de `a x ignore(t) x répand(t)`. Le flux provenant de `connaît` le fait diminuer de `a x répand(t) x (n - ignore(t) - 1)`. Il en résulte que, tout calcul fait :

```
répand(t+Δt) = répand(t) - a x répand(t) x (n - ignore(t) - 1)
```

Une étude simple montre que, pour `ignore(t)` variant de n à 0,5 x (n - 2), `répand` augmente et diminue ensuite pour atteindre la valeur 0.

Exercice 7 : Simulation dynamique : propagation d'une rumeur

On suppose que, initialement, `ignore(0) = n - 1` et `connaît(0) = 0`. On choisit « au hasard » la personne de la population qui commence à répandre la rumeur.

1. Écrire un algorithme qui calcule périodiquement les effectifs de chaque classe de la population et le rapport `ignore / répand`.

Pour ceux qui programment un ordinateur :

2. Réaliser les simulations avec les données suivantes :

```
n =  100 ; a = 10^-4 ;
n = 1000 ; a = 10^-4 ;
```

© ENI Editions - All rigths reserved

`n = 100 ; a = 10^-3 ;`
`n = 10 ; a = 10^-3 .`

On considère cette fois les individus répartis dans une file d'attente dans laquelle les seules rencontres possibles sont celles entre personnes suffisamment proches. On dira qu'une rencontre est possible entre les individus x et y lorsque `|x - y| ≤ e` *où* `e` *est le seuil de rencontres fixé d'avance.*

3. Écrire l'algorithme qui réalise cette simulation.

Les individus d'une population de `n x n` *personnes sont répartis dans un carré. Les seules rencontres qui peuvent se produire sont celles entre deux personnes (x1, y1) et (x2, y2) dont la distance vérifie :*

`|x1 - x2| + |y1 - y2| ≤ e`

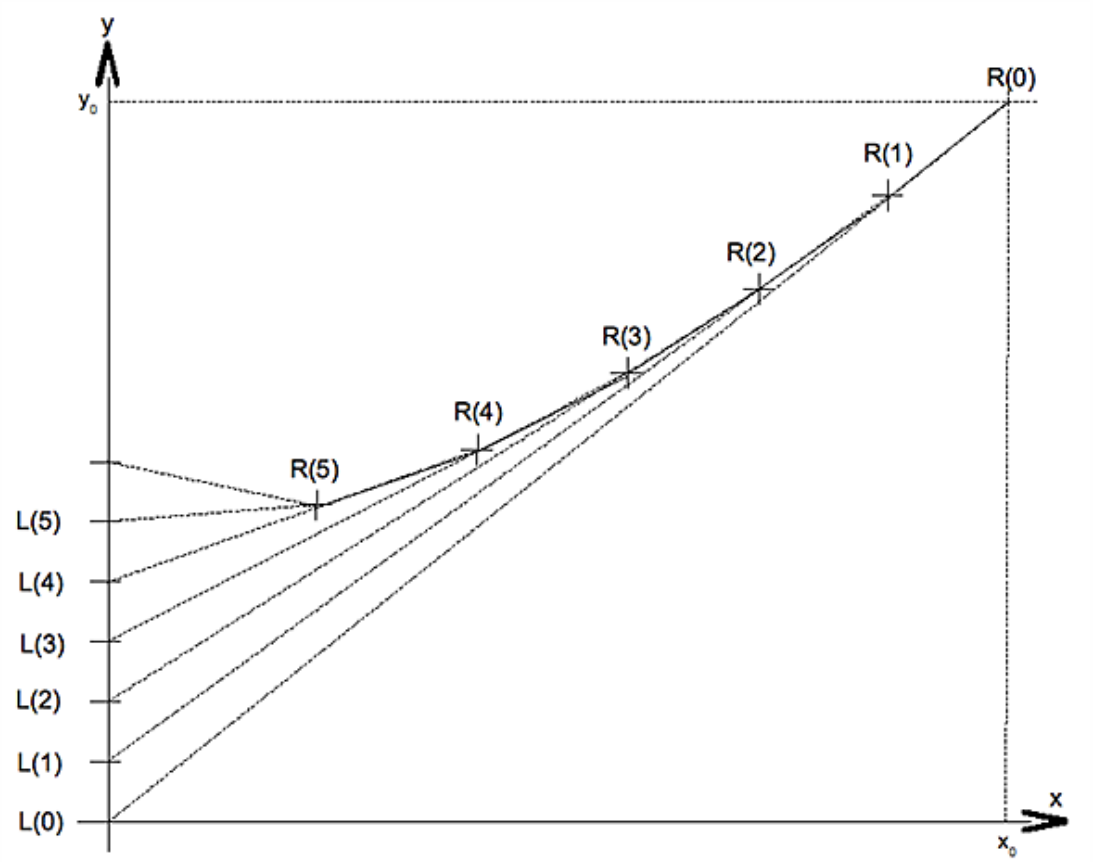
On décide aussi du nombre de personnes répandant initialement la rumeur et de leur position dans la population.

4. Écrire l'algorithme qui réalise cette simulation.

2. Course poursuite

Un renard poursuit un lapin qui tente de lui échapper. Ils se déplacent par bonds successifs de longueur constante. On simplifie encore la simulation en supposant que chacun bondit à son tour, en alternance. De plus, les changements de direction ne se font que lors de la fin d'un bond.

Le lapin suit une trajectoire rectiligne, sur l'axe des ordonnées du repère cartésien du plan dont l'origine est sa position initiale, à `t = 0`. Les seules positions intéressantes sont celles des deux animaux avant ou après un bond. La position du lapin après le saut numéro i, pour i = 0, 1, 2… est le point `L(i)` de coordonnées `L(i) = (0 ; z_i)`. Celle du renard est le point `R(i) = (x_i ; y_i)`. En chaque point `R(i)` depuis sa position initiale `R(0)`, le renard bondit d'une longueur constante `l(R)` en direction du lapin. Le lapin bondit d'une longueur constante `l(L)` sur l'axe des ordonnées. La figure ci-dessous représente quelques pas de la simulation.



La simulation consiste à tracer les trajectoires respectives du lapin et du renard jusqu'à ce que celui-ci capture celui-là.

Exercice résolu 2 : Course poursuite renard - lapin

© ENI Editions - All rigths reserved

Écrire la simulation de la course poursuite entre le renard et le lapin.

Solution partielle

Le chapitre Structures élémentaires a déjà abordé les algorithmes de dessin. Les types **POINT** et **SEGMENT** y ont été définis. Ils permettent des tracés élémentaires. Ici, il reste à montrer comment calculer les positions respectives des deux animaux après chaque saut. Chaque point $L(i)_{i \geq 0}$ a pour coordonnées $(0 ; z_i)_{i \geq 0}$ et chaque point $R(i)_{i \geq 0}$ a pour coordonnées $(x_i ; y_i)_{i \geq 0}$. Après le saut `i - 1`, la distance qui sépare le lapin du renard est donnée par :

$$d_{i-1}=\sqrt{x_{i-1}^2+(z_{i-1}-y_{i-1})^2}$$

Il n'est pas difficile de démontrer les formules suivantes :

$$\begin{cases} x_i=\dfrac{d_{i-1}-l_R}{d_{i-1}}\times x_{i-1} \\ y_i=\dfrac{d_{i-1}-l_R}{d_{i-1}}\times y_{i-1}+l_R\times z_{i-1} \\ z_i=z_{i-1}+l_L \end{cases}$$

Soit **poursuite** la procédure qui réalise les tracés de cette simulation. Sa signature est :

```
poursuite
    (
        R, L   : POINT # Positions initiales du renard et du lapin.
        LR, LL : RÉEL  # Longueur de leurs sauts respectifs.
    )
```

L'analyse de premier niveau conduit à l'algorithme suivant :

Algorithme 6 : Poursuite du renard et du lapin - Version 1.0

```
...
Calculer la distance d entre R et L.
# R et L sont les positions courantes du renard et du lapin
# respectivement.
# Le renard fait un bond de R à R1 puis occupe R1.
Calculer R1
Tracer segment(R, R1)
R <- R1
# Le lapin fait un bond de L à L1 puis occupe L1.
Calculer L1
Tracer segment(L, L1)
L <- L1
...
```

Cette suite de calculs est à répéter jusqu'à ce que le renard rejoigne le lapin, ou qu'il est établi qu'il ne le rejoindra jamais. Le renard l'attrape lorsque la distance qui les sépare est inférieure à la longueur d'un saut : `d < l(R)`. Il ne le rejoindra jamais si la distance qui les sépare augmente. Le reste n'est pas difficile.

Une simulation sur le même principe, moins cruelle mais bien plus belliqueuse, consiste à tracer les trajectoires respectives d'un avion et d'un missile.

Exercice 8 : Poursuite missile - avion

Un avion vole horizontalement à vitesse constante `v_a` à une altitude `h`. À l'instant `t = 0`, un missile est tiré à l'origine du repère en direction de l'avion. Il vole à une vitesse constante `v_m`. L'avion est touché lorsque le missile pénètre dans une sphère de rayon `r` centrée sur l'avion.

© ENI Editions - All rigths reserved

On suppose d'abord que la trajectoire du missile est rectiligne, selon un angle $\propto$ par rapport à l'horizontale et que cette trajectoire ne peut plus être corrigée lorsque le missile est parti.

1. Écrire les algorithmes qui permettent de tracer les trajectoires de l'avion et du missile.

On suppose, à présent, que le missile est capable de corriger automatiquement sa trajectoire pour se diriger vers l'avion.

2. Modifier la solution précédente en conséquence.

© ENI Editions - All rigths reserved

Simulation statistique de phénomènes déterministes

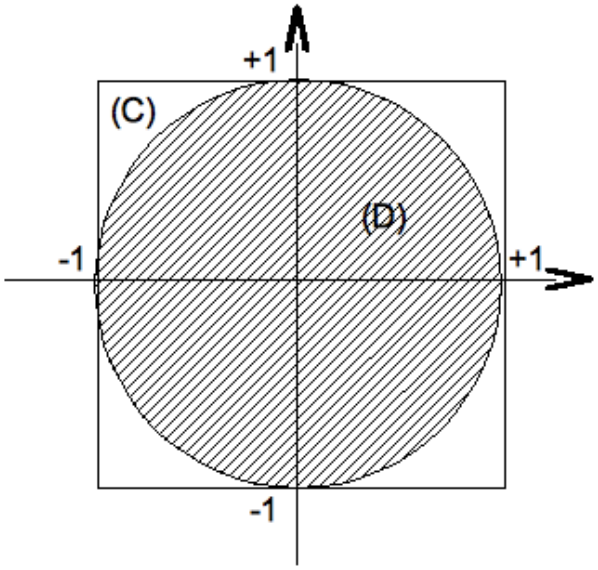
Dans cette section, nous proposons des exercices d'écritures d'algorithmes qui montrent comment estimer des résultats parfaitement déterministes en utilisant des nombres pseudo-aléatoires. Nous cherchons donc à obtenir une estimation statistique de résultats qui ne dépendent pas du hasard. Les méthodes mises en œuvre relèvent de méthodes dites de *Monte-Carlo* en référence à l'univers des jeux de hasard qui ont fait la réputation de la ville du même nom. Ces méthodes sont apparues comme outils de recherche pendant la seconde guerre mondiale afin de simuler le comportement des neutrons dans les matériaux fissiles. Elles sont employées, par exemple, pour calculer des intégrales multiples dites intégrales de configuration.

La section Calculer π montre comment calculer une estimation du nombre π. La méthode utilisée est un cas particulier de calcul d'intégrales définies, présenté à la section Évaluer une intégrale définie.

1. Calculer π

On donne un disque (D) de rayon r. Nous savons calculer son aire. C'est A(D) = π x r^2. Mais *combien vaut π* ? Cette section montre comment obtenir une évaluation statistique de la valeur de π.

π est l'aire d'un disque de rayon r = 1. Ce disque est inscrit dans un carré (C) de côté c = 2 et dont l'aire est A(C) = 4. La figure ci-dessous représente le disque et le carré dans lequel il est inscrit.



On choisit « au hasard » un grand nombre de points, disons N = 10000, dans le carré. Parmi ces `N` points, `n` « tombent » dans le disque. Comme les points sont équiprobables, on a que le nombre de points dans le disque est proportionnel à l'aire du disque. Autrement dit, on a :

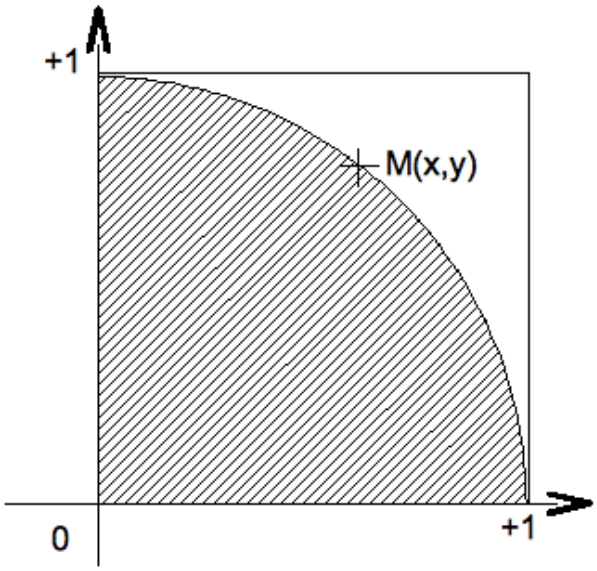
```
A(D) / A(C) = n / N => A(D) = (n / N) x A(C)
```

Posons <π> l'estimation de π. Comme A(D) = π et A(C) = 4, on obtient :

```
<π> = 4 x (n / N)
```

Par conséquent, le rapport `n / N` est une estimation de `π / 4`. Mais `π / 4` est l'aire du quart de disque de rayon `r = 1`, comme le montre la figure suivante :

© ENI Editions - All rigths reserved



Il est donc équivalent de choisir les points « au hasard » dans le carré de côté `c = 1` pour obtenir une estimation de `π / 4`. L'estimation <π> est alors obtenue par :

```
<π> = n / (N / 4)
```

Ainsi, pour N = 10000 points, on obtient <π> = n / 2500.

Exercice 9 : Estimation statistique de π

Choisir un point M « au hasard » dans le carré de côté c = 1 consiste à générer deux réels aléatoires de [0, 1] pour ses coordonnées x et y. Le point obtenu est dans le disque si et seulement si $x^2 + y^2 \leq 1$.

1. Écrire les algorithmes qui permettent d'estimer π statistiquement.

Remarquez que M est dans le quart de disque lorsque ***entier****($x^2 + y^2$) = 0. Il est à l'extérieur du quart de disque lorsque la partie entière vaut 1. On peut éviter les tests en remarquant que pour chaque point M généré on a :*

```
n <- n + 1 - entier( x² + y²)
```

2. Modifier les algorithmes obtenus à la question précédente pour tenir compte de cette remarque.

2. Évaluer une intégrale définie

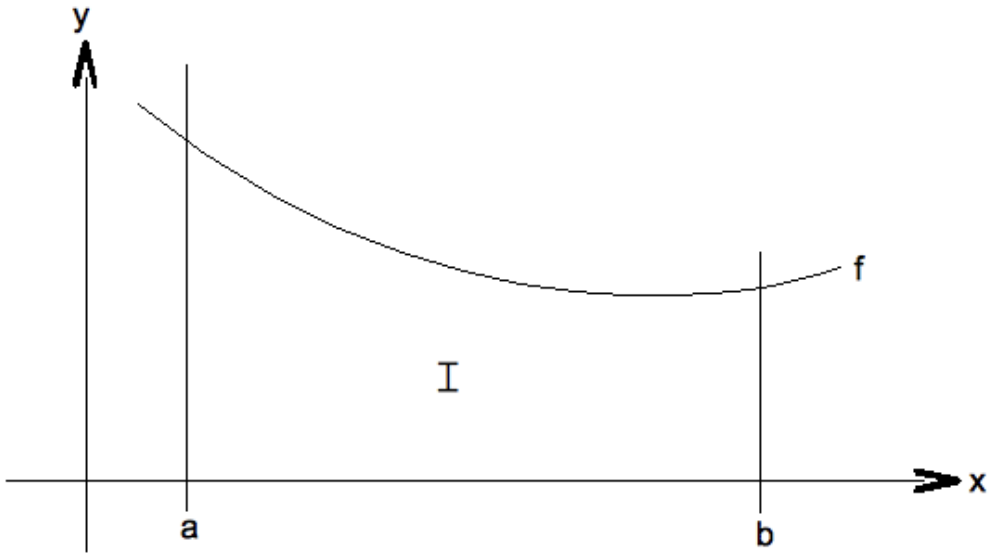
Cette section est largement inspirée de [PEL97].

On donne une fonction f définie sur un intervalle [a, b]. On demande d'estimer statistiquement le nombre `I` donné par :

$$I = \int_a^b f(x).dx$$

On sait que ce problème revient à évaluer une aire. La figure ci-dessous illustre la situation.

© ENI Editions - All rigths reserved



`I` est donc l'aire dont la frontière est la partie du graphe de `f` entre `f(a)` et `f(b)`, les droites `x = a` et `x = b` et enfin l'axe des abscisses. Soit alors $(x_i)_{1 \leq i \leq n}$ une suite de réels aléatoires uniformément répartis sur l'intervalle [a, b]. On démontre que, lorsque `n` est suffisamment grand :

$$\hat{I} = \frac{b-a}{n}\sum_{i=1}^{i=n} f(x_i)$$

Nous savons générer des nombres aléatoires distribués uniformément sur un intervalle donné. Par conséquent, évaluer `Î` ne présente pas de difficulté. L'algorithme de principe est le suivant :

```
...
i     <- 1 # Rang de la prochaine évaluation.
somme <- 0 # f(x1)+f(x2)+...+f(xi).
jusqu'à
    i > n
répéter
    générer un réel aléatoire x dans [a, b]
    somme <- somme + f(x)
    i <- i + 1
fin répéter
Î <- (b - a)/n x somme
...
```

Exercice 10 : Évaluation statistique d'intégrales

Utiliser la méthode exposée ci-dessus pour évaluer :

$$I_1 = \int_0^1 \sqrt{1-x^2}\,dx$$

$$I_2 = \frac{1}{2\pi}\int_0^5 e^{-\frac{x^2}{2}}\,dx$$

$$I_3 = \int_{0,2}^{0,8} e^{\arcsin(x)} \times \frac{x}{\sqrt{1-x^2}}\,dx$$

Le calcul de I_1 est en fait celui de $\pi / 4$. I_2 est égal à 0,5 et $I_3 \approx 0{,}729\,635\,856$.

© ENI Editions - All rigths reserved

Simulation de phénomènes aléatoires

1. À la chasse aux mouches

Cette section pourrait s'intituler : *le hasard au service d'une névrose !*

On veut attraper des mouches pour les enfermer dans un bocal. Les raisons d'un tel comportement ne concernent pas ce livre : dressage, expérimentation scientifique, etc. À chaque capture, il faut ouvrir le couvercle du bocal pour enfermer la nouvelle mouche. Mais alors, chacune des mouches prisonnières, y compris la nouvelle, peut s'échapper, disons avec la probabilité p. *Combien de mouches faut-il attraper, en moyenne, pour obtenir un nombre n de prisonnières ?* Il s'agit de simuler un certain nombre de telles parties de chasses pour obtenir une estimation statistique du nombre moyen de mouches à attraper pour constituer une collection de n exemplaires.

Il est clair que la simulation, dans la mesure où elle fournit une estimation fiable du résultat attendu, est la seule pratique expérimentale possible. Bien entendu, le problème est suffisamment simple pour être résolu exactement par le calcul, mais ce n'est pas notre propos : nous voulons écrire des algorithmes de simulations.

Exercice résolu 3 : Simuler une chasse aux mouches

Pour simuler une capture, il suffit de faire tourner la roulette à deux issues qui sort l'issue 1 avec la probabilité p et l'issue 0 avec la probabilité (1 - p). La roulette est lancée pour chaque mouche prisonnière afin de déterminer si elle s'échappe ou non. Évidemment, elle s'échappe lorsque la roulette sort 1.

Simuler N chasses pour capturer n mouches au cours de chacune d'elles. Estimer le nombre moyen de mouches à capturer.

Solution

On veut simuler la capture de n mouches. Soit **chasse** l'algorithme qui simule une partie de chasse. Il doit connaître le nombre n de mouches à capturer et la probabilité p d'évasion d'une mouche capturée. Il retourne le nombre total de mouches capturées pour en obtenir n. C'est donc une fonction dont la spécification est donnée par l'algorithme suivant.

*Algorithme 7 : Spécification de la fonction **chasse***

```
Algorithme chasse
    # Une chasse pour capturer n mouches. Chacune peut s'évader avec
    # la probabilité p. Rend le nombre total de mouches capturées.
Entrée
    n : ENTIER # Le nombre de mouches attendues.
    p : RÉEL   # La probabilité d'évasion d'une mouche.
Résultat : ENTIER
précondition
    n ≥ 0
    0 ≤ p < 1
postcondition
    Résultat = nombre total de captures
fin chasse
```

La précondition impose que p reste strictement inférieur à 1. En effet, une probabilité d'évasion de 1 conduirait à une simulation de durée infinie.

L'analyse de l'itération est simple.

Faire une hypothèse sur un état intermédiaire

Dans un état intermédiaire, un certain nombre de mouches, disons i, sont enfermées dans le bocal. Pour en arriver là, il a fallu capturer des mouches dont certaines ont réussi à s'échapper. Soit **Résultat** le nombre de mouches capturées pour faire i prisonnières.

Hypothèse (H) : i mouches sont enfermées dans le bocal. **Résultat** est le nombre de captures réalisées.

Remarquez qu'il n'est pas possible d'écrire **Résultat** = **chasse**(i, p). En effet, le résultat renvoyé par la fonction **chasse** est aléatoire et rien ne dit qu'un appel à cette fonction avec les arguments i et p donnera la valeur actuelle de **Résultat**. Cette valeur est obtenue « au hasard ».

© ENI Editions - All rigths reserved

Voir si c'est fini

C'est fini lorsque `i = n`. Là encore, on ne peut pas utiliser `n - i`, le nombre de mouches qui doivent encore être capturées, comme variant de contrôle. Ce nombre est bien la distance à la solution, mais ce n'est pas un variant de contrôle de l'itération car il ne décroît pas vers 0 d'une façon monotone à cause des évasions. En fait, cette distance prend des valeurs aléatoires et elle converge vers 0 statistiquement. Étudier le problème sérieusement dépasse les objectifs de ce livre.

Se rapprocher de l'état final

Lorsque ce n'est pas fini, on se rapproche de l'état final en capturant une nouvelle mouche et en l'enfermant dans le bocal :

```
capturer une nouvelle mouche : Résultat <- Résultat + 1
introduire la mouche dans le bocal et ajuster i
recommencer
```

Initialiser le calcul

L'état initial correspond à un bocal vide et aucune mouche n'a encore été enfermée :

```
...
initialisation
    Résultat <- 0 # Nombre de captures réalisées.
    i <- 0        # Nombre de mouches dans le bocal.
...
```

Rédiger l'algorithme définitif

Nous obtenons l'algorithme ci-dessous.

*Algorithme 8 : Définition de la fonction **chasse***

```
Algorithme chasse
    # Une chasse pour capturer n mouches. Chacune peut s'évader
    # avec la probabilité p. Rend le nombre total de mouches
    # capturées.
Entrée
    n : ENTIER # Le nombre de mouches attendues.
    p : RÉEL   # La probabilité d'évasion d'une mouche.
Résultat : ENTIER
précondition
    n ≥ 0
    0 ≤ p < 1
variable
    i : ENTIER
initialisation
    Résultat <- 0 # Nombre de captures réalisées.
    i <- 0        # Nombre de mouches dans le bocal.
jusqu'à
    i = n
répéter
    # Capturer une nouvelle mouche.Résultat <- Résultat + 1
    # L'introduire dans le bocal.
    i <- i - introduction(i + 1, p) + 1
fin répéter
postcondition
    Résultat = nombre total de captures
fin chasse
```

La fonction **introduction** simule l'introduction d'une mouche dans le bocal. Elle calcule et retourne le nombre de mouches qui parviennent à s'échapper. Pour cela, elle utilise la roulette a deux issues déjà présentée. Pour chaque mouche prisonnière, l'algorithme fait tourner la roulette. La mouche s'évade lorsque la roulette sort l'issue 1 ; soit **ROULETTE** le type de données permettant d'implanter le générateur. L'algorithme suivant définit la fonction

© ENI Editions - All rigths reserved

introduction.

Algorithme 9 : Introduction d'une mouche dans le bocal

```
Algorithme introduction
    # Introduit un mouche dans le bocal qui en contient déjà n - 1.
    # Chacune s'échappe avec la probabilité p. Retourne le nombre
    # de mouches qui s'échappent.
Entrée
    n : ENTIER    # Nombre de mouches à obtenir.
    p : RÉEL      # Probabilité d'évasion d'une mouche.
Résultat : ENTIER# Nombre d'évasions
précondition
    n ≥ 0
    0 ≤ p < 1
variable
    i : ENTIER # Nombre de mouches traitées.
    r : ROULETTE
initialisationRésultat <- 0 # Nombre d'évasions.
    i <- 0        #  Nombre de mouches traitées.
    # Initialiser le générateur de la roulette.
    r.p <- p
    initialiser(r)
jusqu'à
    i > n
invariant
    ...
variant de contrôle
    n - i + 1
répéter
    suivant(r)
    Résultat <- Résultat + item(r)
    i <- i + 1
fin répéter
postcondition
    Résultat = nombre d'évasions
fin introduction
```

Cette fonction calcule, pour chaque mouche, si elle s'échappe ou non. Elle retourne le nombre d'évasions, c'est-à-dire le nombre d'issues égales à 1 dans une suite de n essais. Lorsque n est exactement le nombre de captures attendues, c'est-à-dire lorsque le bocal contient déjà n - 1 mouches, la chasse se termine si **introduction** retourne un résultat nul, donc si la roulette utilisée produit n issues toutes égales à 0. Ceci suggère un autre exercice de simulation : *quelle est la longueur moyenne d'une suite d'issues identiques dans un ensemble de tirages de la roulette ?* En fait, c'est ce problème qui est « habillé » en une chasse aux mouches.

Le problème suivant, appelé problème de Banach, met en jeu un fumeur de cigarettes (fumer tue, mais nous simulons !). Ce problème se résout comme le précédent.

Exercice 11 : Les allumettes de Banach

Un fumeur possède deux boîtes de d'allumettes, l'une dans la poche droite et l'autre dans la poche gauche de sa veste. La boîte dans la poche droite contient d allumettes et celle dans la poche gauche en contient g. Lorsqu'il veut allumer une cigarette, il choisit une poche « au hasard » et allume sa cigarette. Combien reste-t-il d'allumettes dans l'autre boîte lorsqu'il constate que la boîte qu'il a choisie est vide ?

1. Écrire les algorithmes de simulation lorsqu'il choisit l'une ou l'autre poche avec la même probabilité.

2. Même question, mais le fumeur est gaucher et choisit donc la poche gauche plus souvent que la poche droite.

La deuxième question doit être précisée et il faut d'abord donner un sens à l'expression « [il] choisit donc la poche gauche plus souvent que la poche droite ».

2. Propagation d'une rumeur

La section Simulation de processus dynamiques a montré comment étudier l'évolution d'une population d'individus dans laquelle circule une rumeur. On connaissait les lois qui régissent la circulation de l'information dans la population.

© ENI Editions - All rigths reserved

La simulation a consisté à appliquer mécaniquement ces règles d'évolution pour obtenir les états successifs de la population. Le mécanisme d'évolution était parfaitement déterminé par les règles appliquées et la simulation a consisté à accélérer la diffusion de la rumeur. Dans cette section, nous reprenons la même étude, mais cette fois en appliquant les règles d'évolution à des individus qui entrent en interaction « au hasard ». Concrètement, on génère deux entiers aléatoires qui identifient deux individus de la population et on applique les règles d'évolution de la simulation dynamique.

Exercice résolu 4 : Propagation d'une rumeur - Simulation aléatoire

On s'intéresse au nombre de rencontres nécessaires pour que la rumeur s'éteigne et aux effectifs des différents groupes de population lorsqu'elle disparaît.

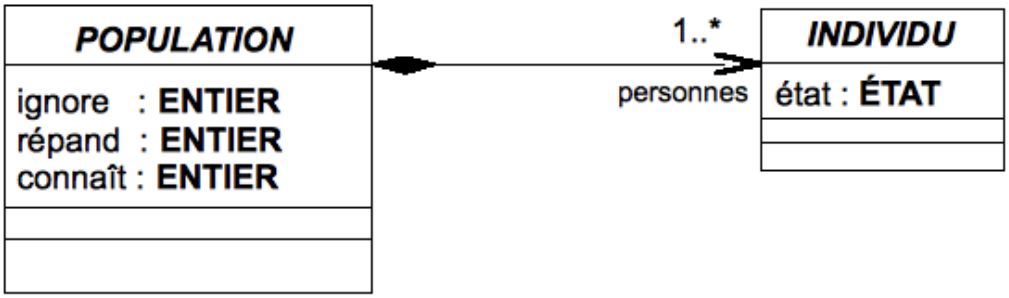
Écrire les algorithmes permettant de réaliser cette simulation.

Solution

On considère donc une population composée d'individus identiques. Chaque individu est caractérisé par un état capturé par la valeur d'une variable *état* du type **ÉTAT**, défini par l'énumération :

```
type
    ÉTAT
défini par l'énumération
    {ignore, répand, connaît}
fin ÉTAT
```

L'état global de la population est capturé par trois compteurs entiers dont la valeur est le nombre d'individus de la population dans chacun des états du type **ÉTAT**. La figure ci-dessous est un schéma conceptuel qui montre l'association entre une population et les individus qui la composent.



Une population n'existe pas indépendamment des individus qui la composent ; d'où le losange noir que nous avons déjà rencontré. Une population est composée d'un ou de plusieurs individus. Les instances d'**INDIVIDUs** qui composent une population donnée forment un ensemble, une collection, *personnes* de **PERSONNEs**. Un individu donné participe à la composition d'une population unique, du moins dans cet exemple. La multiplicité 1 du côté population est implicite. Cependant, ce schéma ne caractérise pas encore entièrement les propriétés de cette population. En effet, l'état de la population est représenté par les valeurs des variables du schéma, mais ces variables ne sont pas quelconques. En toutes circonstances, à tout instant, la somme des effectifs des trois groupes d'individus est égale à la taille de la population, c'est-à-dire au nombre de *personnes* qui la composent. Soit **cardinal** la fonction entière qui rend le nombre d'éléments de la collection qu'elle reçoit en paramètre. Ici, ce sera *personnes*. L'entité **POPULATION** est caractérisée par l'invariant :

```
invariant
    ignore + répand + connaît = cardinal(personnes)
```

On suppose donc que l'effectif de la population reste invariable pendant la durée de la simulation. L'invariant devrait être précisé dans la dernière case du schéma qui représente l'entité **POPULATION**.

Nous pouvons déjà définir le type **INDIVIDU** :

```
type
    INDIVIDU
structure
    état : ÉTAT
fin INDIVIDU
```

Pour définir le type **POPULATION**, il faut décider de la structure de données à utiliser pour implanter les personnes. Nous n'avons pas le choix : nous n'en connaissons que deux, la chaîne de caractères et le tableau. Utilisons un tableau :

© ENI Editions - All rigths reserved

```
type
    POPULATION
structure
    personnes : TABLEAU[INDIVIDU][1,n]
    ignore    : ENTIER
    répand    : ENTIER
    connaît   : ENTIER
invariant
    ignore + répand + connaît = cardinal(personnes)
    cardinal(personnes) ≤ n
fin POPULATION
```

La simulation consiste à initialiser la population en choisissant une personne qui commence à répandre la rumeur. Nous simulons alors des rencontres entre deux personnes en générant aléatoirement deux entiers entre 1 et **cardinal** (personnes). Ces entiers sont les numéros des cases des personnes qui se rencontrent dans le tableau des individus. On applique alors les règles d'évolution utilisées lors de la simulation dynamique.

L'algorithme principal initialise la simulation puis organise les rencontres en surveillant le nombre des personnes qui répandent la rumeur.

```
Initialiser la simulation
jusqu'à
    un nombre nul de personnes répandant la rumeur
répéter
    calculer une rencontre
    mettre à jour l'état de la population
    compter une rencontre de plus
fin répéter
...
```

Soit **ALÉA_ENTIER**, le type de données utilisées par le générateur de nombres aléatoires pour produire des entiers uniformément répartis sur un intervalle donné [a, b]. Autrement dit, chacun des entiers de l'intervalle a la probabilité 1 / (b - a + 1) d'être tiré. Le générateur est initialisé en lui communiquant les bornes a et b. On obtient l'algorithme ci-dessous.

Algorithme 10 : Propagation d'une rumeur - Simulation aléatoire

```
Algorithme propagation_rumeur
    # Simule la propagation jusqu'à l'extinction de la rumeur.
constante
    n : ENTIER <- 10000 # Taille de la population.
variable
    p    : POPULATION
    i, j : ENTIER# Indices des individus qui se rencontrent.
    nbRencontres : ENTIER# Nombre de rencontres simulées.
    a    : ALÉA_ENTIER    # Structure utilisée par le générateur.
initialisation
    # Réalisation de l'invariant.
    p.répand <- 1 ; p.ignore <- n - 1 ; p.connaît <- 0
    # Création du générateur d'entiers dans [1, n].
    initialiser(a, 1, n)
    # Initialement, personne ne connaît la rumeur.
    Initialiser la population à l'état ignore
    # Déterminer la personne qui commence à répandre la rumeur.
    suivant(a) ; i <- item(a) ; p.personnes[i].état <- répand
    # Encore aucune rencontre.
    nbRencontres <- 0
jusqu'à
    p.répand = 0
répéter# Indices des personnes qui se rencontrent.
    suivant(a) ; i <- item(a)
    suivant(a) ; j <- item(a)
    # Calcul de la rencontre.
    rencontre(p, cardinal(p.personnes), i, j)
    nbRencontres <- nbRencontres + 1
fin répéter
```

© ENI Editions - All rigths reserved

```
...
fin propagation_rumeur
```

La procédure **rencontre** applique les règles d'évolution utilisées dans la simulation dynamique pour mettre à jour l'état de la population. Elle est définie par l'algorithme ci-dessous.

*Algorithme 11 : Définition de la procédure **rencontre***

```
Algorithme rencontre
    # Organiser la rencontre des individus de numéros i et j.
Entrée
    p    : POPULATION
    n    : ENTIER # L'effectif de la population.
    i, j : ENTIER # Les numéros des individus qui se rencontrent.
précondition
    index_min(p.personnes) ≤ i ≤ index_min(p.personnes) + n - 1
    index_min(p.personnes) ≤ j ≤ index_min(p.personnes) + n - 1
    index_min(p.personnes) + n - 1 ≤ index_max(p.personnes)
réalisation
    si
        (
            p.personnes[i].état = ignore
          et
            p.personnes[j].état = répand
        )
      ou
        (
            p.personnes[i].état = répand
          et
            p.personnes[j].état = ignore
        )
    alors
        # Rencontre répand -- ignore => les deux répandent.
        p.ignore <- p.ignore - 1
        p.répand <- p.répand + 1
        p.personnes[i].état <- répand
        p.personnes[j].état <- répand
    sinon si
        (
            p.personnes[i].état = répand
          et
            p.personnes[j].état = connaît
        )
      ou
        (
            p.personnes[i].état = connaît
          et
            p.personnes[j].état = répand
        )
    alors
        # Rencontre répand -- connaît => les deux connaissent.
        p.répand <- p.répand - 1
        p.connaît <- p.connaît + 1
        p.personnes[i].état <- connaît
        p.personnes[j].état <- connaît
    sinon si
        (
            p.personnes[i].état = répand
          et
            p.personnes[j].état = répand
        )
    alors
        # Rencontre répand -- répand => les deux connaissent et
        # aucun ne répand plus.
        p.répand <- p.répand - 2
        p.connaît <- p.connaît + 2
        p.personnes[i].état <- connaît
```


© ENI Editions - All rigths reserved

```
        p.personnes[j].état <- connaît
    fin si
postcondition
    # n, i et j ne sont pas modifiés.
    n = ancien(n)
    i = ancien(i)
    j = ancien(j)
    # Règles appliquées lors des rencontres.
    ignore -- répand  => répand  -- répand
    répand -- répand  => connaît -- connaît
    répand -- connaît => connaît -- connaît
fin rencontre
```

La postcondition est rédigée rapidement. Son expression complète est difficile. Elle est laissée en exercice.

Exercice 12 : Propagation d'une rumeur - Autre version

La solution précédente a utilisé un tableau pour implanter l'ensemble des personnes qui composent la population. On peut, à la place, utiliser une chaîne de caractères. La population est une chaîne dans laquelle chaque caractère représente l'état d'un individu. Il répand la rumeur lorsque le caractère est R ; il l'ignore si le caractère est I ; il la connaît quand le caractère est C. La chaîne est donc composée des seuls caractères R, I et C.

Écrire les algorithmes en utilisant une chaîne de caractères pour représenter la population dans laquelle circule la rumeur.

L'exercice suivant n'est pas différent de la propagation d'une rumeur.

Exercice 13 : Propagation d'un incendie

L'exercice consiste à simuler la propagation d'un incendie dans une forêt. On suppose que les arbres sont répartis aux nœuds d'un réseau rectangulaire. À l'instant initial, un arbre prend feu, par exemple à la suite d'une imprudence. Il communique le feu à ses voisins selon des règles de propagation à préciser.

1. Écrire les algorithmes de simulation de la propagation du feu.

On cherche à répondre à des questions simples :

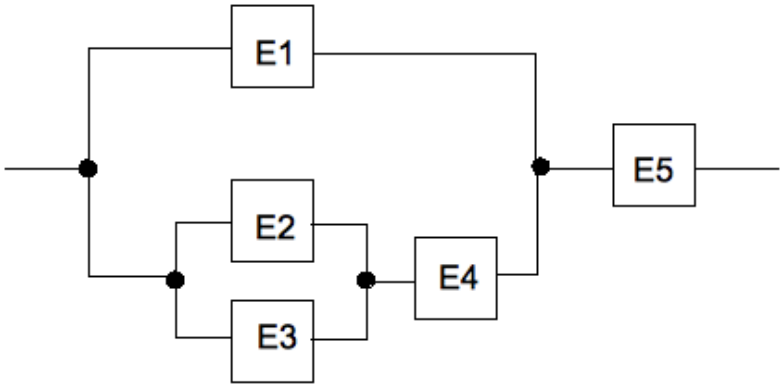
- *combien d'arbres sont brûlés lorsque le feu s'éteint ? ;*
- *comment faire intervenir dans la simulation la distance entre les arbres ? ;*
- *existe-t-il une répartition des arbres qui minimise le nombre de sujets brûlés ? ;*

2. Reprendre la simulation en tenant compte de la direction et de la force du vent.

3. Reprendre la simulation en répartissant les arbres « au hasard » dans le rectangle.

3. Fiabilité des systèmes

Un composant électronique est fait de cinq éléments interconnectés selon le schéma de la figure ci-dessous.



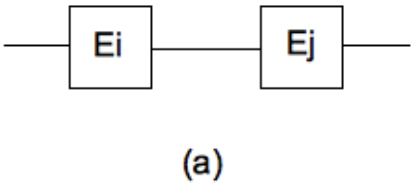
© ENI Editions - All rigths reserved

Chacun des éléments E_i peut tomber en panne, indépendamment des autres éléments. On veut estimer la probabilité de bon fonctionnement du composant quand on connaît la probabilité p_i d'un élément E_i de fonctionner 1000 heures sans panne.

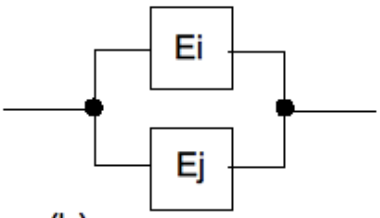
Soient p_i et p_j, les probabilités de bon fonctionnement des éléments E_i et E_j après 1000 heures. La probabilité de bon fonctionnement d'un assemblage de ces deux éléments dépend du type d'assemblage :

- E_i et E_j sont montés en parallèle, comme sur la figure (b) ci-dessous. La probabilité que les deux éléments tombent en panne simultanément est $p_i \times p_j$ et la probabilité de bon fonctionnement de l'ensemble est $p = p_i + p_j - p_i \times p_j$;

- E_i et E_j sont montés en série, comme sur la figure (a) ci-dessous. La probabilité de bon fonctionnement est $p = p_i \times p_j$.



(a)



(b)

Exercice 14 : Fiabilité d'un composant électronique

Écrire les algorithmes permettant d'estimer par simulation la probabilité de bon fonctionnement du composant de la figure précédente après 1000 heures.

On utilise encore la roulette introduite dans une section précédente pour obtenir une issue $e_i \in \{0, 1\}$ pour chaque élément E_i du circuit. Le circuit fonctionne encore après 1000 heures si C = 1 avec :

$$C = \{[(e_2 + e_3) \times e_4] + e_1\} \times e_5$$

On simule ainsi n circuits C_1, C_2, …, C_n. La probabilité cherchée est estimée par la formule : $p = (C_1 + C_2 + \ldots + C_n) / n$. Le calcul exact de cette probabilité donne $p \approx 0{,}62$ avec $p = 0{,}7$, $i = 1, 2, \ldots, 5$.

4. Dispersion des valeurs des composants d'un circuit électronique

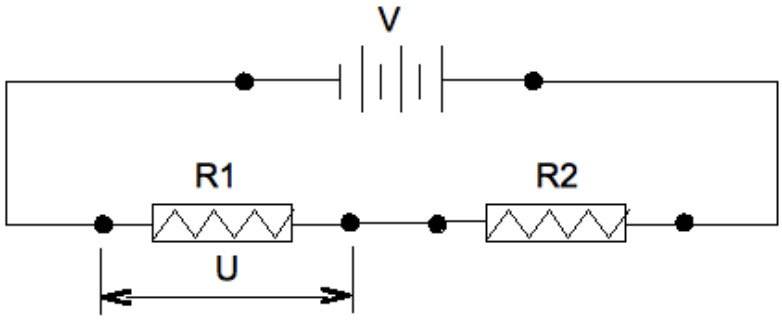
Les caractéristiques, c'est-à-dire les valeurs de la réponse, d'un circuit électronique sont déterminées par la valeur des composants qui réalisent le circuit. Ces caractéristiques sont calculées en attribuant à chaque composant une valeur précise et fixée, ce que ne garantissent pas les fabricants. En fait, le fabriquant d'un composant garantit une tolérance sur la valeur nominale du composant. Ainsi, par exemple, telle résistance au carbone a une valeur nominale de R = 3 kΩ avec une tolérance de 10 % sur cette valeur, soit une résistance R telle que 2700 Ω ≤ R ≤ 3300 Ω. Ainsi, le domaine des valeurs possibles de R est l'intervalle [2700, 3300] en Ω. La réalisation industrielle d'un circuit se fait donc à partir de composants dont la valeur réelle s'écarte de la valeur théorique prévue par le calcul. Il s'agit alors de s'assurer que cette dispersion des caractéristiques des composants n'a pas de conséquences qui rendraient les circuits fabriqués inutilisables.

Le but de cette section est d'étudier l'influence de la dispersion des caractéristiques des composants sur la réponse d'un circuit à partir d'un exemple élémentaire.

Exercice résolu 5 : Réponse d'un diviseur de tension

Considérons le circuit de la figure suivante.

© ENI Editions - All rigths reserved



Les résistances R1 et R2 sont montées en diviseur de tension. On démontre que la tension U aux bornes de la résistance R1 est donnée par la formule : U = V x (R1 / (R1+R2)).

Écrire les algorithmes permettant de donner une évaluation statistique de la réponse en tension du diviseur en fonction de la dispersion des valeurs des résistances.

On s'intéresse à la valeur moyenne de U, l'écart type et l'histogramme.

Solution

La simulation consiste à choisir aléatoirement les valeurs des résistances R1 et R2 pour calculer U en supposant un générateur parfait délivrant une tension V fixe et précise. On procède à n simulations pour en déduire les résultats demandés. L'histogramme consiste à diviser l'intervalle dans lequel se répartissent les valeurs de U en un nombre fixé k d'intervalles et à dénombrer les valeurs de U qui appartiennent à chacun de ces intervalles élémentaires.

Exemple

Posons V = 1,5 V, R1 = R2 = 3 kΩ. Comme les deux résistances ont des valeurs nominales égales, la formule précédente donne U = 0,75 V. Supposons que la tolérance sur les valeurs des résistances est de 10 %. Chacune d'elles a donc une valeur comprise entre 2700 Ω et 3300 Ω. Un calcul simple montre alors que U prend sa valeur dans un intervalle de longueur L = 0,15 V centré sur la valeur nominale U = 0,75 V.

Soit k = 10 le nombre d'intervalles élémentaires pour représenter la distribution des valeurs de U par un histogramme de 10 intervalles. Ces intervalles ont une largeur de l / 10 = 0,015 V. Les bornes de ces intervalles sont donc :

0,675 ; 0,690 ; 0,705 ; 0,720 ; 0,735 ; 0,750 ; 0,765 ; 0,780 ; 0,795 ; 0,810 ; 0,825

La fabrication de lots produit des composants dont les valeurs se répartissent autour de la valeur nominale suivant une loi normale (courbe « en cloche »). Pour simplifier, on considère ici que cette répartition est rectangulaire, c'est-à-dire uniforme dans l'intervalle de tolérance indiqué par le fabricant. Les instructions qui permettent le calcul de U sont alors :

```
...
r1 <- hasard(r1, r1min, r1max)
r2 <- hasard(r2, r2min, r2max)
u <- v x (1 + r2 / r1)
...
```

Ici, **hasard** est une fonction qui génère un réel aléatoire à partir de son premier argument selon une distribution rectangulaire entre les valeurs de ses deux derniers arguments. Ces derniers sont les valeurs extrêmes des résistances calculées à partir de la valeur nominale et de la tolérance sur cette valeur.

Cette base permet de simuler n circuits pour obtenir une estimation Û de U et les indicateurs statistiques demandés.

Chaque résistance a une valeur nominale Ri qui vérifie : Rimin ≤ Ri ≤ Rimax. On démontre que la tension U vérifie Umin ≤ U ≤ Umax avec :

Umin = V x (R1min / (R1max+R2max))

Umax = V x (R1max / (R1min+R2min))

L'amplitude L est donc : L = Umax - Umin.

La tension U va prendre des valeurs dans l'intervalle [Umin, Umax]. On les répartit dans k > 0 classes pour réaliser l'histogramme. Chaque classe a une longueur l = L / k. Les frontières de ces classes sont :

U1 = Umin

U2 = U1 + l = Umin + l

U3 = U2 + l = Umin + 2 x l

© ENI Editions - All rigths reserved

...

Uk = Umin + (k - 1) x l

Les frontières des classes sont donc données par :

U1 = Umin

Uj = Umin + (j - 1) x l pour 2 ≤ j ≤ k+1

Il reste à comprendre comment placer une valeur Û de U obtenue par simulation. Soit histogramme un tableau dont les cases reçoivent les fréquences absolues des valeurs de Û selon les classes précalculées. Ainsi, par exemple, histogramme[3] reçoit la fréquence des valeurs de Û comprises dans [U3, U4]. Étant donnée une valeur Û, le numéro de la case de histogramme qui doit être incrémentée est :

```
numCase <- entier((Û - U) / l) + 1
```

On obtient ainsi une valeur de l'intervalle dont la fréquence est enregistrée dans la case de numéro numCase. La mise à jour de cette fréquence consiste à écrire :

```
histogramme[numCase] <- histogramme[numCase] + 1
```

L'essentiel des calculs est donc établi par les formules qui précèdent. Il reste à choisir les structures de données qui permettront ces calculs. On définit d'abord un type **DIVISEUR** qui regroupe les paramètres d'un diviseur de tension :

```
type
    DIVISEUR
structure
    V : RÉEL # Différence de potentiel aux bornes du générateur.
    R1, R2   : RÉEL # Valeurs nominales des résistances.
    tR1, tR2 : RÉEL # Tolérances en % sur valeurs des résistances.
fin DIVISEUR
```

Les résultats sont la moyenne des valeurs obtenues pour Û, l'écart type et le tableau des fréquences absolues de ces valeurs réparties en k classes. Cependant, l'utilisateur des services de cet algorithme doit aussi connaître les valeurs des frontières des classes calculées. Tous ces résultats sont regroupés dans une structure de données de type **RÉSULTATS** défini par :

```
type
    RÉSULTATS
structure
    moyenne    : RÉEL   # Moyenne des valeurs de Û.
    écart_type : RÉEL   # Écart type sur les valeurs de Û.
    nbClasses  : ENTIER # Nombre de classes.
    # Frontières des classes.
    classes    : TABLEAU[RÉEL][1, nbClasses + 1]
    # Fréquences de chaque classe.
    histogramme: TABLEAU[RÉEL][1, nbClasses]
fin RÉSULTATS
```

Soit **diviseurDeTension** la fonction qui réalise les calculs demandés. Sa signature est :

```
diviseurDeTension
        (
            d : DIVISEUR # paramètres du diviseur.
            N : # Nombre de simulations.
        ) : RÉSULTATS
```

Pour le reste, les calculs sont élémentaires et ils sont laissés en exercices.

© ENI Editions - All rigths reserved

Notes bibliographiques

Le 147-générateur et le test du poker sont inspirés du petit livre [RAD77] qui propose près de 150 exercices parfois passionnants et dont les solutions sont commentées. Les tests des générateurs sont étudiés d'une façon approfondie par [KNU73a].

© ENI Editions - All rigths reserved

Résumé

Ce chapitre a proposé quelques activités simples permettant de simuler des processus déterministes ou aléatoires. Les formules utilisées peuvent paraître parfois impressionnantes, mais le contenu mathématique reste d'un niveau élémentaire. Par contre, les algorithmes proposés sont souvent difficiles à spécifier formellement et les vérifications et les tests sont d'autant plus difficiles que les résultats sont imprévisibles. Tous les problèmes posés se résolvent aisément par le calcul et il est ainsi possible de vérifier la cohérence des résultats obtenus informatiquement avec ceux donnés par les calculs mathématiques.

© ENI Editions - All rigths reserved

Bibliographie

[RAD77] Lennart RADE : *Tentez votre chance avec votre calculateur programmable* ; CEDIC, 1977.

[KNU73a] Donald KNUTH : *The Art of Computer Programming - Vol. 1* ; ADDISON WESLEY, 1973.

© ENI Editions - All rigths reserved

Introduction

On s'intéresse à la *sécurité* des données d'un système informatique. C'est un problème vaste, difficile, qui revêt de multiples aspects. Ici, nous étudions et mettons en pratique d'une façon simplifiée quelques-uns de ces aspects.

La deuxième section étudie comment assurer *l'intégrité* des données. Il s'agit de vérifier que les données mises en œuvre n'ont pas subi de modification. La section suivante aborde les techniques qui assurent la *confidentialité* des données. On veut ainsi ne permettre l'accès aux données qu'à un nombre restreint d'entités utilisatrices bien caractérisées. Nous devrions ensuite nous intéresser aux méthodes pour *identifier*, *authentifier* et *autoriser* les entités qui accèdent aux données mais ce livre n'aborde pas ces aspects du problème.

© ENI Editions - All rigths reserved

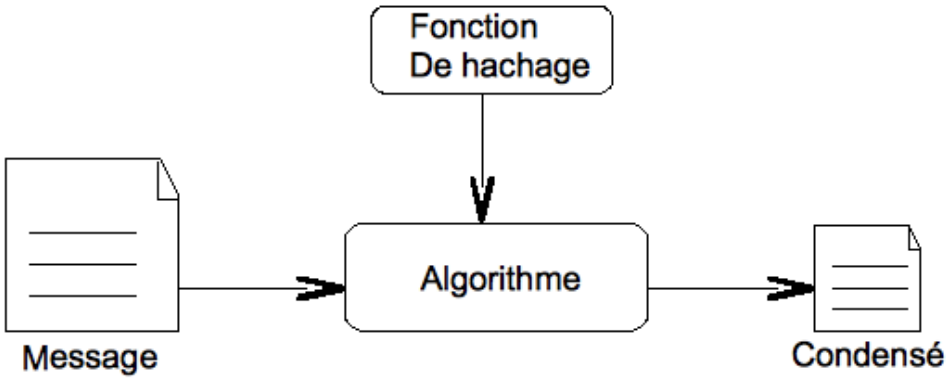
Intégrité

1. Présentation

Les données auxquelles nous nous intéressons « contiennent » une certaine « information ». Toute perturbation des données va modifier cette information.

Les perturbations peuvent être de différentes natures. Ainsi, par exemple, lors d'écritures ou de lectures vers ou depuis un disque, l'environnement électromagnétique du système peut modifier les données d'une façon imprévisible pendant leur transfert. De même, pendant un échange de données sur un réseau, un utilisateur peut intercepter et modifier les messages d'une façon illégitime ou inattendue. Une application peut corrompre les données sur lesquelles elle travaille, à la suite d'un incident de fonctionnement par exemple.

Les méthodes usuelles auxquelles nous nous intéressons dans cette section permettent de s'assurer que les données conservent « l'information qu'elles contiennent ». Nous voulons ainsi nous convaincre que les données n'ont pas subi de modification. Il s'agit donc de s'assurer de *l'intégrité* des données utilisées. Le principe général d'une telle méthode est illustré par la figure ci-dessous.



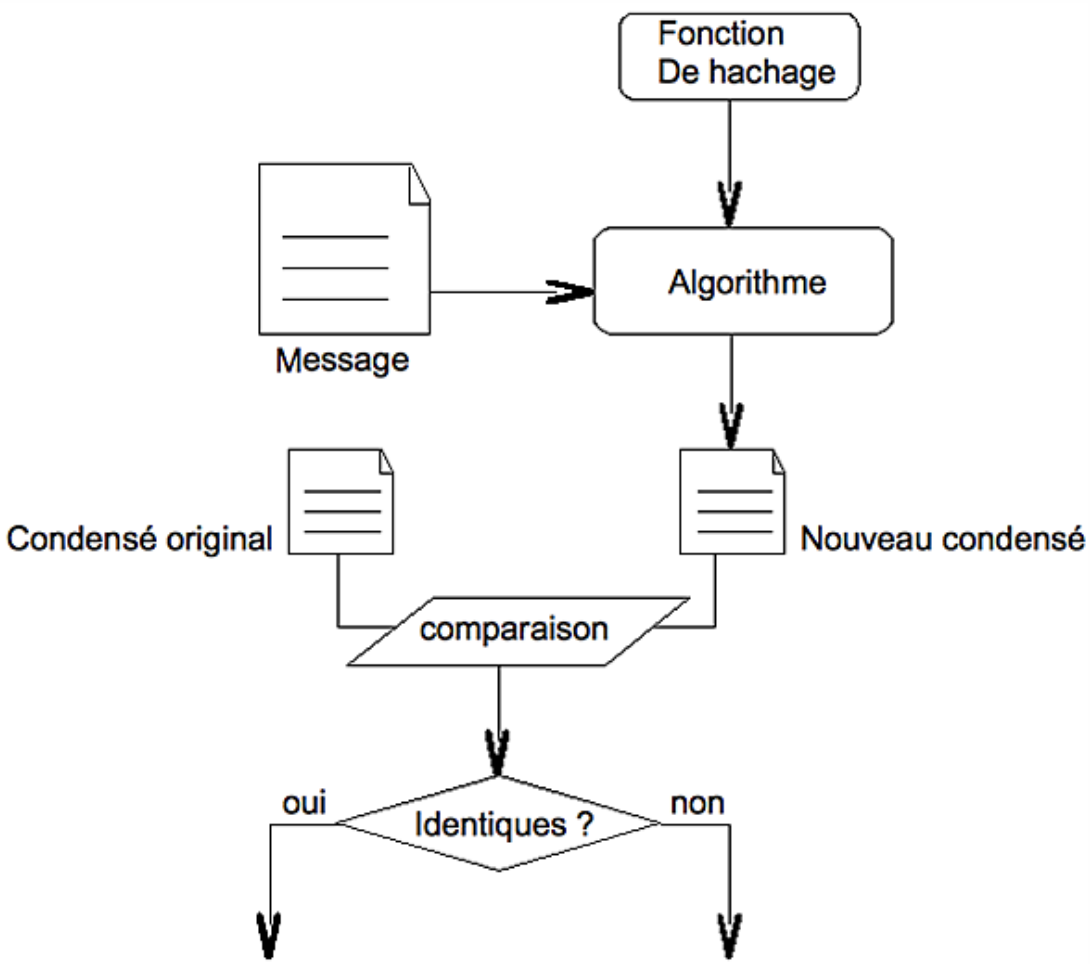
Un algorithme calcule un *condensé* ou *empreinte* (*hash*) du message. Quelle que soit la taille du message, le condensé aura une longueur fixe, par exemple de 32 octets (256 bits). Ainsi, toute l'information contenue dans le message sera comprimée, « condensée » en 32 octets.

Pour s'assurer que les données n'ont pas été perturbées, on recalcule, avec le même algorithme, le condensé du message à vérifier et on le compare au condensé original. Le détail des opérations est donc :

- recalculer le nouveau condensé ;
- comparer le nouveau condensé au condensé original ;
- s'ils sont différents, c'est que les données ou le condensé original ont subi des modifications.

Cette méthode est illustrée par la figure suivante.

© ENI Editions - All rigths reserved



Sur cette figure, le condensé original est le condensé envoyé par l'émetteur et reçu par son destinataire avec le message. Le nouveau condensé est l'empreinte calculée par le récepteur du message. On prétend que le message n'a pas subi de modification lorsque le nouveau condensé calculé est identique au condensé reçu. Cependant, il est clair que le nombre d'empreintes distinctes est fini, exactement de 2^{256} pour une empreinte de 32 octets, alors que le nombre de messages à condenser est potentiellement illimité. Il existe donc de nombreux messages ayant la même empreinte. Ainsi, alors que l'empreinte anthropométrique identifie un individu d'une manière unique, il n'en est pas de même de l'empreinte informatique. Par conséquent, la « quantité d'informations » d'un condensé n'est pas la même que celle du texte qu'il condense. On appelle « collision » le fait d'obtenir la même empreinte pour plusieurs messages distincts. On diminue la probabilité de collision en augmentant le nombre de bits de l'empreinte.

Il existe de nombreux algorithmes de hachage. Les plus connus sont MD5 et SHA-1. Il n'est pas question d'entrer dans les détails algorithmiques des fonctions qui permettent de calculer une empreinte. Dans la suite, nous considérons que nous disposons d'une fonction **hachage** qui permet de calculer l'empreinte d'une chaîne de caractères selon une fonction mathématique et un algorithme qui ne sont pas précisés. Cette fonction rend une chaîne de caractères de longueur fixe, indépendante de la taille de son argument. Sa spécification est donnée par l'algorithme ci-dessous.

*Algorithme 1 : Spécification de la fonction **hachage***

```
Algorithme hachage
    # L'empreinte de ch.
Entrée
    ch : CHAÎNE # La chaîne de caractères à condenser.
Résultat : CHAÎNE
précondition
    estDéfinie(ch)
postcondition
    Résultat est le condensé de ch
    longueur(Résultat) est une constante donnée quelle que soit ch
fin hachage
```

2. Comparer deux empreintes

© ENI Editions - All rigths reserved

Commençons par un exercice simple.

Exercice résolu 1 : Comparaison de deux empreintes

Soient deux empreintes e1 et e2, obtenues par la fonction hachage définie ci-dessus.

1. Écrire un algorithme qui détermine le nombre de caractères différents de même rang dans ces deux condensés.

2. Montrer comment utiliser ces fonctions pour s'assurer de l'intégrité d'un message.

Solution

On veut compter le nombre de caractères différents aux mêmes rangs des deux empreintes. L'algorithme renvoie un entier et c'est donc une fonction. Les deux condensés à comparer sont nécessairement de même longueur et c'est une précondition. La définition de cette fonction est simple à analyser et à écrire. L'analyse complète est laissée en exercice. L'algorithme ci-dessous en donne la spécification et l'algorithme suivant donne la version définitive de cette définition dans laquelle on suppose que les caractères d'une chaîne sont numérotés à partir de 1.

*Algorithme 2 : Spécification de la fonction **différences***

```
Algorithme différences
    # Nombre de caractères distincts dans les mêmes positions de
    # e1 et e2.
Entrée
    e1, e2 : CHAÎNE # Les chaînes à comparer.
Résultat : ENTIER
précondition
    estDéfinie(e1)
    estDéfinie(e2)
    longueur(e1) = longueur(e2)
postcondition
    longueur(e1) = 0 => Résultat = 0
    longueur(e1) > 0 et premier(e1) ≠ premier(e2) =>
             Résultat = 1 + différences(fin(e1), fin(e2))
    longueur(e1) > 0 et premier(e1) = premier(e2) =>
             Résultat = différences(fin(e1), fin(e2))
fin différences
```

La définition complète est donc la suivante :

*Algorithme 2 : Spécification de la fonction **différences***

```
Algorithme différences
    # Nombre de caractères distincts dans les mêmes positions de
    # e1 et e2.
...
variable
    lgr : ENTIER# Nombre de caractères d'un empreinte.
    i   : ENTIER # Numéro du prochain caractère à observer.
initialisation
    lgr <- longueur(e1)
    Résultat <- 0
    i <- 1
jusqu'à
    i > lgr
        invariant
            i = 1 => Résultat = 0
            i > 1 => Résultat = différences
                (
                    sousChaîne(e1, 1, i - 1),
                    sousChaîne(e2, 1, i - 1)
                )
        variant de contrôle
            lgr - i + 1
```

© ENI Editions - All rigths reserved

```
répéter
    si
        item(e1, i) ≠ item(e2, i)
    alors
        Résultat <- Résultat + 1
    fin si
    assertion
        Résultat = différences
                (
                    sousChaîne(e1, 1, i),
                    sousChaîne(e2, 1, i)
                )
    i <- i + 1
    assertion
        Résultat = différences
                (
                    sousChaîne(e1, 1, i - 1),
                    sousChaîne(e2, 1, i - 1)
                )
fin répéter
...
fin différences
```

La fonction **sousChaîne** retourne une copie de son premier argument qui commence au caractère dont le numéro est la valeur du second argument et qui termine au caractère dont le numéro est la valeur du troisième argument.

On a reçu un message *message* et son empreinte *empreinte* dont on veut s'assurer de l'intégrité. Les opérations à réaliser sont :

```
...
# calcul du condensé du message reçu.
condensé <- hachage(message)
# Calcul du nombre de caractères différents.
d <- différences(empreinte, condensé)
# Message intègre ssi différences = 0.
si
    d ≠ 0
alors
    le message ou son empreinte a (ont) été modifié(s)
sinon
    pas de modification, ni du message ni de l'empreinte
fin si
...
```

Remarquez que le calcul du nombre de caractères différents n'est pas nécessaire ici. En effet, il suffit que les deux chaînes *empreinte* et *condensé* soient différentes au sens de la non égalité des chaînes de caractères. Il suffit donc d'écrire :

```
si
    empreinte ≠ condensé
alors
    le message ou son empreinte a (ont) été modifié(s)
sinon
    pas de modification, ni du message ni de l'empreinte
fin si
...
```

puisque l'opérateur ≠ est applicable aux chaînes de caractères. Sinon, il suffirait de le définir.

Un *message* n'est pas toujours la valeur d'une variable. En fait, « message » est à prendre au sens large de « chaîne de caractères » dont la longueur est quelconque, arbitrairement grande, et dont chaque caractère n'est pas nécessairement éditable. Nous devons donc considérer, dans le cas général, que *message* est une donnée binaire et, le plus souvent, cette « donnée » est enregistrée dans un fichier. La section suivante traite du calcul du condensé d'un fichier.

3. Condenser le contenu d'un fichier

© ENI Editions - All rigths reserved

Soit un message à envoyer à un correspondant. Il s'agit d'un fichier à accès séquentiel, enregistré sur un support qui n'est pas adressable, contenant des caractères éditables. Nous voulons joindre à ce fichier un nouveau fichier binaire à préparer qui contiendra l'empreinte du fichier texte. Ce qui distingue cet exercice des précédents, c'est que le fichier à condenser est arbitrairement grand. Nous ne pouvons donc pas lire la totalité du contenu du fichier dans une chaîne de caractères unique ch pour écrire ensuite :

```
empreinte <- hachage(ch)
```

Nous supposons donc que la fonction de hachage ne peut traiter des chaînes de caractères de longueur arbitraire.

Exercice résolu 2 : Condenser un fichier

Écrire l'algorithme qui condense un fichier texte à organisation et accès séquentiel.

Solution partielle

Pour réussir à calculer une empreinte d'un fichier de longueur arbitraire, nous pouvons le condenser par bloc. La méthode consiste à découper le contenu du fichier en blocs de taille fixe et à condenser chaque bloc. Cependant, nous ne pouvons pas transmettre les empreintes de tous les blocs. En fait, on exige d'obtenir une empreinte unique, de la taille habituelle d'une empreinte retournée par la fonction de hachage. Précisons le problème.

On se donne un fichier *lettre.txt* qui est un fichier à organisation et accès séquentiel, contenant des caractères éditables. Nous voulons calculer un condensé, pour l'enregistrer ensuite dans un fichier binaire *lettre.hach*. Le fichier est découpé en blocs de taille fixe puis chaque bloc, augmenté du condensé du bloc précédent, est lui-même condensé. Cependant, pour que les blocs restent de taille fixe, on concatène le condensé obtenu pour un bloc avec le bloc suivant dont la taille est ajustée pour que le bloc obtenu soit de la taille fixe choisie. Ainsi, par exemple, supposons que la taille choisie pour les blocs soit de 4Ko, soit 4096 octets. Lorsque la taille d'une empreinte est de 32 octets, on a :

```
...
lire dans le fichier lettre.txt un bloc de 4064 octets
concaténer ce bloc avec l'empreinte obtenue pour le bloc précédent
calculer l'empreinte de ce nouveau bloc
...
```

L'empreinte définitive sera la dernière empreinte calculée. Remarquer que la taille du bloc qui est condensé est 4064 + 32 = 4096 octets. Par conséquent, la taille d'un bloc lu depuis le fichier est donnée par :

```
TAILLE_LUE = TAILLE_BLOC - TAILLE_EMPREINTE
```

Soit donc **condenserFichier**, la fonction qui retourne le condensé d'un fichier dont elle reçoit, en argument, le nom interne. L'analyse de premier niveau de cette fonction est la suivante :

*Algorithme 4 : Analyse de premier niveau de **condenserFichier***

```
Algorithme condenserFichier
    # Condenser le contenu d'un fichier.
...
Ouvrir le fichier à condenser en mode LECTURE BINAIRE
Positionner le curseur devant le premier bloc
Initialiser le condensé à CHAÎNE_VIDE
Lire un bloc de longueur L depuis le fichier
tant que
    la fin du fichier n'est pas atteinte
répéter
    Concaténer le bloc lu avec le condensé actuel
    # On obtient ainsi un nouveau bloc de longueur L.
    Condenser le nouveau bloc obtenu
    Avancer au bloc suivant dans le fichier
    Lire un bloc de longueur L - TAILLE_EMPREINTE
fin répéter
si
    la taille du dernier bloc lu > 0
alors
    Concaténer le bloc lu avec le condensé actuel
    Condenser le nouveau bloc obtenu
fin si
```

© ENI Editions - All rigths reserved

```
Fermer le fichier d'entrée
...
```

Remarquez le traitement particulier pour le dernier bloc. En effet, une dernière lecture tente une lecture au delà de la marque de fin de fichier. Par conséquent, la fonction **finDeFichier** retourne alors VRAI et le dernier bloc lu, peut-être de taille inférieure à la taille standard d'un bloc, n'est pas traité dans l'itération. Il doit donc être traité indépendamment des autres blocs. Cependant, l'avant dernière lecture peut avoir amené un bloc de la bonne longueur, dont le dernier caractère serait exactement le caractère qui précède la marque de fin de fichier. Dans ce cas, après l'avant-dernière lecture, **finDeFichier** rend encore FAUX et la lecture suivante renvoie un bloc vide ; c'est ce qui justifie le test sur la longueur du dernier bloc lu.

Cette fonction est alors utilisée de la façon suivante :

```
...
constante
    # Nombre d'octets d'un bloc à condenser et du condensé.
    TAILLE_BLOC      : ENTIER <- 4096
    TAILLE_EMPREINTE : ENTIER <- 32
    # Taille d'un bloc lu depuis le fichier.
    TAILLE_LUE : ENTIER <- TAILLE_BLOC - TAILLE_EMPREINTE
    # Nom des fichiers dans le SGF.
    nom     : NOM_EXTERNE <- 'lettre.txt'
    hachage : NOM_EXTERNE <- 'lettre.hach'
variable
    empreinte : CHAÎNE # Le condensé calculé.
    fichier : NOM_INTERNE # Descripteur du fichier à hacher.
réalisation
    empreinte <- condenserFichier(nom)
                 ouvrir(hachage, 'ÉCRITURE_BINAIRE')
    fichier <- descripteur(hachage)
    début(fichier)
    écrire(fichier, empreinte)
    fermer(fichier)
...
```

Le cœur de la fonction **condenserFichier** consiste à lire un bloc dans le fichier, le concaténer à l'empreinte déjà obtenue et à condenser le résultat de cette concaténation. Les instructions de base sont donc :

```
...
bloc <- bloc ⊕ empreinte # Concaténation.
empreinte <- hachage(bloc)
bloc <- lire(fichier, TAILLE_LUE)
...
```

La lecture d'initialisation doit copier depuis le fichier un bloc de TAILLE_BLOC octets puisque, dans cet état, l'empreinte est vide et sa longueur est nulle.

Il reste à écrire la version définitive et complète de la fonction. Elle ne présente plus de difficulté et sa réalisation est laissée en exercice.

Exercice 1 : Condenser un fichier

*1. Écrire la version définitive de l'algorithme de la fonction **condenserFichier**.*

2. Écrire l'algorithme qui prend en entrée un fichier et son empreinte. Il recalcule le condensé du fichier et le compare à l'empreinte reçue.

© ENI Editions - All rigths reserved

Confidentialité

On cherche, cette fois, à s'assurer que les seules entités qui pourront prendre connaissance des données sont celles prévues. Le principe consiste à masquer les données afin qu'elles ne soient plus directement compréhensibles par toute autre entité que l'entité destinataire. On parle alors de *cryptage* des données.

1. Principes mathématiques de la confidentialité

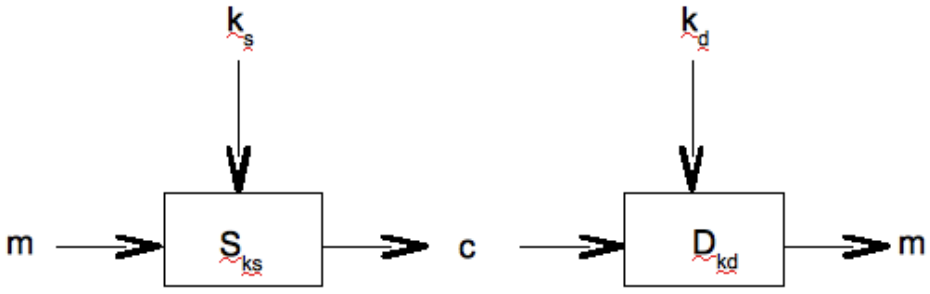
Soient M l'ensemble des messages en clair, C l'ensemble des messages cryptés et K l'ensemble des clés de chiffrement. L'opération de chiffrement est une fonction S de M x K dans C :

$$S : M \times K \rightarrow C$$
$$(m, k_s) \hookrightarrow S_{ks}(m) = c$$

Le déchiffrement est une fonction D de C x K dans M :

$$D : C \times K \rightarrow M$$
$$(c, k_d) \hookrightarrow D_{kd}(c) = m$$

Les clés de chiffrement/déchiffrement sont k_s et k_d respectivement. Les deux fonctions vérifient : $D_{kd}(S_{ks}(m)) = m$. Cette formule exprime que le chiffrement est réversible. On peut représenter ces opérations comme sur le dessin de la figure ci-dessous.



Les algorithmes de chiffrement utilisent différentes relations mutuelles entre k_s et k_d.

2. Cryptographie à clé secrète

Lorsqu'il existe un moyen simple d'obtenir k_d à partir de k_s, par exemple lorsque $k_d = k_s$, on parle de *chiffrement symétrique* ou encore de *chiffrement à clé secrète.* Dans ce cas, les deux extrémités partagent un secret commun. Ce peut être, par exemple, la même valeur de clé $k = k_s = k_d$. Toute entité qui connaît k peut alors déchiffrer le message c pour obtenir le message m.

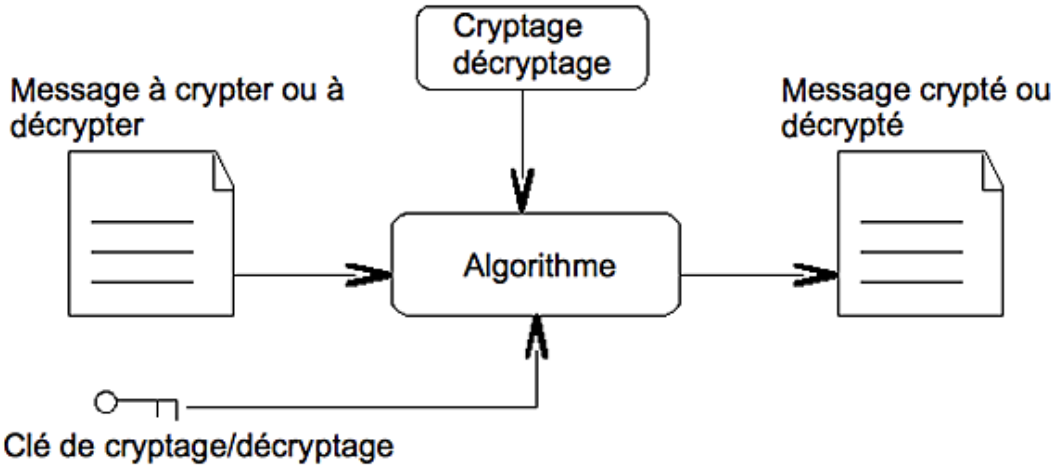
On sait, en théorie, assurer une *confidentialité parfaite* avec ce type de cryptage où les entités qui communiquent partagent un secret commun. L'algorithme de VERNAM est particulièrement simple et assure le *secret parfait* des échanges.

Soient A et B, comme Alice et Bob par exemple, les deux entités qui partagent une clé commune k. Cette clé est une *suite aléatoire de bits* de même longueur que le message `m` à envoyer. Pour crypter le message, A utilise la fonction S_k = `m` ⊗ k, dans laquelle ⊗ désigne l'opérateur booléen OU EXCLUSIF. Cet opérateur est usuellement noté ⊕, mais nous avons utilisé ce dernier symbole pour noter la concaténation des chaînes de caractères et nous continuerons à le faire. Comme la fonction S_k est une involution, B utilise D_k = c ⊗ k pour retrouver `m`. Le secret parfait exige que le nombre de clés soit aussi grand que le nombre de messages en clair possibles, ce qui explique que la clé doit être aussi longue, en nombre de bits, que le message. De plus, elle doit être *parfaitement aléatoire*, ce qui est une condition difficile à satisfaire. Enfin, A et B doivent échanger cette clé d'une façon sûre, par un canal fiable.

© ENI Editions - All rigths reserved

3. Cryptographie à clé symétrique

C'est la méthode déduite des considérations de la section précédente. On utilise une donnée particulière appelée la *clé de cryptage* pour crypter les données du message. La clé de cryptage est connue de l'émetteur et du destinataire du message et on parle alors de cryptage à clé symétrique. Le cryptage fait subir aux données une modification à l'aide d'un algorithme qui utilise la clé. L'algorithme applique une fonction qui est réversible. Par conséquent, l'application de cette même fonction aux données cryptées permet de retrouver le message original. La confidentialité recherchée n'est donc obtenue que si la clé n'est connue que des entités prévues. La principe de la méthode est résumé par le schéma de la figure ci-dessous.



Les paragraphes suivants demandent de mettre en pratique quelques algorithmes élémentaires de cryptage à clé symétrique.

4. Codage élémentaire : XOR

La clé est une chaîne de caractères de longueur l arbitraire. Les données à crypter sont découpées en blocs de longueur l. Chaque bloc est crypté en lui faisant subir une opération « ou exclusif » (XOR) avec la clé.

Exercice résolu 3 : Cryptage symétrique XOR

On note ⊗ l'opérateur XOR.

1. Rappeler la table de vérité de ⊗. Montrer que XOR est une opération involutive.

Soit `lettre.txt`, un fichier texte à organisation et accès séquentiels.

2. Écrire l'algorithme qui prend en entrée le nom du fichier à crypter et la clé de cryptage et qui écrit sur le disque le fichier crypté.

3. Justifier l'utilisation de la même clé pour crypter et décrypter le fichier. En déduire que le même algorithme sert aussi à décrypter le fichier.

4. Quels sont les principaux inconvénients de cette méthode ?

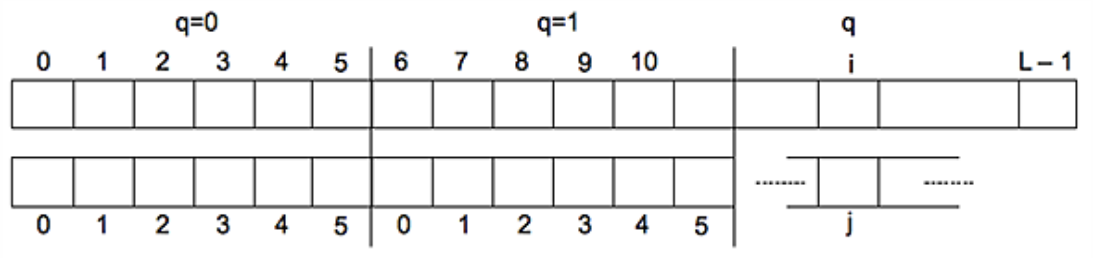
On suppose que le message à crypter n'est plus de longueur arbitraire, mais de longueur assez réduite pour tenir tout entier dans une variable.

5. Refaire cet algorithme, mais cette fois, il génère aléatoirement une clé aussi longue que le message à crypter.

Solution partielle

On ne résout ici que la question 2 en ce qui concerne les calculs d'indices des caractères pour un message noté *phrase* de longueur L. Supposons d'abord que les caractères de la phrase et de la clé sont numérotés à partir de 0. La figure ci-dessous représente la mise en correspondance d'une phrase de L caractères et d'une clé de l = 6 caractères.

© ENI Editions - All rigths reserved



Soit i le numéro d'un caractère quelconque de la phrase et j le caractère qui lui correspond dans la clé. *Quelle relation existe-t-il entre i et j ?*

La clé « découpe » la phrase en un ensemble de blocs, tous de même longueur l , sauf peut-être le dernier. Soit q le numéro du bloc qui contient i avec q = 0 le numéro du premier bloc. Il n'est pas difficile de constater que i = q x l + j avec 0 ≤ j < l. Par conséquent, j est le reste de la division de i par l alors que le numéro du bloc qui contient i est le quotient q de cette division. Les caractères d'une chaîne sont numérotés consécutivement, mais à partir d'un numéro quelconque pour le premier caractère. Soit alors début le numéro du premier caractère à crypter dans la phrase et **index_min**(clé) le numéro du premier caractère de la clé. La formule précédente doit être utilisée avec une translation de i et j pour tenir compte de cette numérotation. On a donc :

```
début ≤ i ≤ fin => (i - début) ≥ 0
```

D'où :

```
0 ≤ (i - début) ≤ fin - début
```

Par conséquent :

```
(i - début) = q x l + (j - index_min(clé))
```

Finalement, on obtient pour le calcul de j :

```
j = reste(i - début, l) + index_min(clé)
```

Exemple

*Soient l = 6, i = 13, début = 1 et **index_min**(clé) = 1. On a alors i - début = 12 et **reste** (12, 6) = 0. Par conséquent, j = 0 + 1 = 1.*

Le cryptage d'un caractère s'organise alors de la façon suivante :

```
...
# Récupérer un caractère de phrase.
carP <- item(phrase, i)
# Calculer le numéro du caractère correspondant de la clé.
j <- reste(i - début, l) + index_min(clé)
# Récupérer le caractère de la clé.
carC <- item(clé, j)
# Calculer le caractère crypté.
c <- carP ⊗ carC
...
```

Le reste de l'exercice ne pose plus de difficulté.

5. Codage de VERNAM

C'est encore la méthode précédente, mais cette fois la clé est aussi longue que le message à crypter. Pour chaque bloc du fichier à crypter, on génère aléatoirement une clé de cryptage de même longueur. Le fichier est crypté et le résultat du chiffrement est enregistré sur disque. Parallèlement, la clé du bloc est aussi enregistrée dans un autre fichier binaire. Le fichier crypté peut emprunter un canal non fiable, mais il est clair que la clé, elle, doit emprunter un canal fiable. Elle ne peut pas être retrouvée par calcul puisqu'elle est, en théorie, parfaitement aléatoire. Cependant, utiliser et transmettre une clé aussi longue que le message est fortement pénalisant. Comme, de plus, on ne peut pas générer une clé parfaitement aléatoire, cette méthode n'est pas utilisée. Nous allons cependant l'analyser pour préparer un algorithme qui chiffre un fichier.

© ENI Editions - All rigths reserved

Exercice résolu 4 : Codage de VERNAM

On donne un fichier texte à organisation et accès séquentiels.

1. Écrire l'algorithme qui chiffre ce fichier dans un fichier binaire et qui sauvegarde la clé aléatoire dans un second fichier binaire.

2. Écrire l'algorithme qui déchiffre un fichier.

Solution partielle

Le principe est identique à celui utilisé pour le calcul de l'empreinte d'un « gros » fichier, partiellement résolu plus haut dans ce chapitre. Comme pour cet algorithme, on ne donne ici que l'analyse de premier niveau. Les choix algorithmiques étant arrêtés, l'écriture de la définition complète ne présente plus de difficulté.

*Algorithme 5 : Procédure **vernam** - Analyse de premier niveau*

```
Algorithme vernam
...
Ouvrir le fichier à crypter (fichier d'entrée) en mode
LECTURE BINAIRE
Ouvrir le fichier de la clé en mode ÉCRITURE BINAIRE
Ouvrir le fichier résultat en mode ÉCRITURE BINAIRE
Positionner le curseur devant le premier bloc
Lire un bloc de longueur TAILLE_BLOC depuis le fichier d'entrée
tant que
    la fin du fichier d'entrée n'est pas atteinte
répéter
    Générer une clé aléatoire de la longueur du bloc
    Crypter le bloc avec la clé
    Enregistrer le bloc crypté avec la clé dans leurs fichiers
    respectifs
    Avancer au bloc suivant dans les fichiers
    Lire un bloc de longueur TAILLE_BLOC depuis le fichier d'entrée
fin répéter
si la taille du dernier bloc lu > 0 alors
    Générer une clé aléatoire de la taille du dernier bloc lu
    Crypter le dernier bloc lu
    Enregistrer le bloc crypté et la clé
fin si
Fermer les trois fichiers
...
```

Il est indispensable d'enregistrer la clé pour pouvoir décrypter le message en la réutilisant. D'autre part, les deux entités qui s'échangent le message crypté n'ont pas besoin de s'entendre sur la longueur du bloc puisque l'opération est une addition bit à bit sans retenue (XOR). Mais comme la clé est aussi longue que le message à crypter, échanger des messages sur un canal fiable n'est pas plus compliqué qu'échanger les clés et, par conséquent, cette méthode n'est pas praticable. On peut, cependant, l'utiliser sans échanger les clés. La méthode consiste à réaliser, à chaque extrémité, un double cryptage alterné en utilisant à chaque fois la même clé, mais qui reste privée et seulement connue de l'extrémité qui l'utilise. L'exercice suivant montre comment procéder.

Voyons d'abord pourquoi l'échange de la clé k n'est pas nécessaire pour le chiffrage XOR.

Exercice 2 : Codage XOR et échange de la clé

Le protocole d'émission du message m de A à B est le suivant :

a. A crypte le message m à l'aide de sa clé k_a qu'il est le seul à connaître et envoie le message crypté c_{ka} à B ;

b. B crypte le message c_{ka} qu'il reçoit de A avec sa clé k_b, qu'il est le seul à connaître, et envoie le résultat C_{ka}^{kb} à A ;

c. A re-crypte C_{ka}^{kb} avec sa clé k_a et envoie le message c obtenu à B ;

d. B crypte c avec sa clé k_b et obtient ainsi m ;

1. Faire un diagramme pour résumer ces opérations.

2. Démontrer que le message en clair obtenu par B à la dernière opération est le message m envoyé par A.

© ENI Editions - All rigths reserved

On voit que ce protocole permet de s'affranchir de la contrainte de l'échange des clés sur un canal qui n'a donc pas à être fiable. Par contre, chaque message fait un aller-retour complet supplémentaire.

6. Codage élémentaire à substitution mono-alphabétique simple

Il s'agit, cette fois, de crypter chaque caractère du fichier, individuellement. Par conséquent, les blocs sont, à présent, de longueur L = 1. La clé est elle-même de longueur 1, mais la clé de décodage n'est plus la même que la clé de cryptage. Cependant, elle s'en déduit par un calcul élémentaire. Il s'agit donc encore d'un algorithme de la famille des algorithmes de cryptage à clé symétrique.

Exercice 3 : Substitution monoalphabétique simple

La clé est un caractère de code clé. Pour chaque caractère à crypter, de code codeCar, on calcule :

```
reste(codeCar + clé, 256)
```

Pour décrypter un caractère, on utilise la clé (256 - clé) et on applique le même algorithme.

Exemple

Soient 'A' de code ASCII 65 le caractère à crypter et le caractère 'H', de code ASCII 72, la clé de cryptage. Le caractère crypté aura le code ASCII 65 + 72 = 137. Pour le décoder, on utilise la clé 256 - 72 = 184. Le décodage donne :

```
reste(137 + 184, 256) = reste(321, 256) = 65 soit 'A'
```

1. Écrire l'algorithme qui réalise le cryptage d'un fichier texte selon cet algorithme.

2. Appliquer cet algorithme pour décoder le fichier obtenu à la question précédente.

7. La méthode de VIGENÈRE

Le cryptage selon la méthode de VIGENÈRE utilise une table de correspondance des caractères et une clé de cryptage secrète. Chaque caractère du message sert à repérer une ligne de la table de correspondance. Chaque caractère de même rang de la clé permet de repérer une colonne de cette table. À l'intersection de la ligne et de la colonne ainsi repérées, on trouve le caractère de substitution du caractère du message.

Exemple

Considérons le message m = 'BABA' et la clé k = 'CACA'. La table de correspondance est arbitraire. En voici une :

	A	***B***	***C***
A	K	R	T
B	E	Z	F
C	I	X	S

Le cryptage procède ainsi :

- *caractère 1 de m = 'B' ; caractère 1 de k = 'C' ; ligne 'B', colonne 'C' donne 'F' ;*
- *caractère 2 de m = 'A' ; caractère 2 de k = 'A' ; ligne 'A', colonne 'A' donne 'K' ;*
- *même codage pour les deux autres caractères.*

Le message codé est donc 'FKFK'.

Lorsque la taille de la clé est inférieure à celle du message, on « répète » la même clé autant de fois que c'est nécessaire :

© ENI Editions - All rigths reserved

```
m = 'Papa, maman, la bonne et moi.'
k = 'CACACACACACACACACACACACACAC'
```

Exercice 4 : Cryptage de VIGENÈRE

Pour simplifier, on considère des messages uniquement constitués de majuscules de l'alphabet, mais sans accent ni ponctuation. La clé est de longueur quelconque et les caractères qui la constituent sont aussi des caractères sans accent.

Dans un premier temps, la table de correspondance est une matrice de 26 lignes et 26 colonnes. La première ligne porte les caractères de 'A' à 'Z'. La seconde, les caractères de 'B' à 'Z' et le dernier est un 'A'. La troisième porte les caractères de 'C' à 'Z' et les deux derniers sont 'A' et 'B'. Ainsi, chaque ligne est une permutation circulaire de la précédente, la première étant faite des 26 majuscules de l'alphabet.

1. Écrire l'algorithme d'initialisation de la table de correspondance.

2. Écrire les algorithmes permettant de coder et de décoder un message.

Comme chaque ligne est une permutation circulaire de la ligne qui la précède, la table de correspondance n'est pas nécessaire. Pour chaque couple de caractères (m, k) extraits du message et de la clé, un calcul à partir de leurs codes permet d'obtenir le code puis le caractère correspondant du message crypté.

3. Écrire les algorithmes qui se déduisent de ces formules.

4. Refaire ces algorithmes lorsque la table de correspondance est obtenue « au hasard ». Mêmes contraintes pour la clé.

5. Étudier le problème lorsqu'on autorise un caractère quelconque, majuscule, minuscule, ponctuation, accent, etc. dans le message à coder et dans la clé de cryptage.

6. Étendre le problème au cryptage d'un fichier quelconque.

Les exercices précédents ont mis en évidence la sensibilité de la clé. Elle doit être disponible pour l'entité qui crypte le message et pour celle qui le décrypte. Elle est le maillon faible de ces méthodes. Il faut donc s'assurer qu'elle reste secrète en toute circonstance. Il existe des méthodes qui permettent aux deux entités de calculer une clé commune sans échanger de données confidentielles, mais ces méthodes dépassent le cadre de cette initiation.

© ENI Editions - All rigths reserved

Notes bibliographiques

L’essentiel de ce chapitre est inspiré de [MAR04]. C’est un livre difficile pour une initiation mais une référence rigoureuse pour une introduction aux techniques de chiffrement. Le protocole d’échange de l’exercice 2 est extrait d’un cours personnel. L’exercice sur le codage monoalphabétique simple est inspiré d’un exercice publié par le CERTA. L’exercice sur le codage de VIGENÈRE est inspiré d’un ancien article paru dans la revue MICRO-SYSTÈMES disparue depuis déjà de nombreuses années.

© ENI Editions - All rigths reserved

Conclusion

La sécurité des données doit assurer leur intégrité, la confidentialité des échanges et l'identification/authentification des entités qui réalisent ces échanges. Ce chapitre a exposé quelques idées simples pour vérifier l'intégrité ou pour assurer la confidentialité des échanges. Les principes ont été présentés à partir de méthodes élémentaires qui ne sont pas utilisables en l'état, mais qui donnent une idée juste de la réalité. Le domaine est cependant bien plus vaste et difficile que ne peut le laisser penser cette présentation, dont le seul objectif était de fournir un contexte pour étudier quelques algorithmes.

© ENI Editions - All rigths reserved

Bibliographie

[MAR04] Bruno MARTIN : *Codage, cryptologie et applications* ; PRESSES POLYTECHNIQUES ET UNIVERSITAIRES ROMANDES, 2004.

© ENI Editions - All rigths reserved